NAVIGATION SOLUTION FOR THE TEXAS A&M AUTONOMOUS GROUND VEHICLE

A Thesis

by

CRAIG ALLEN ODOM

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2006

Major Subject: Mechanical Engineering

NAVIGATION SOLUTION FOR THE TEXAS A&M AUTONOMOUS GROUND VEHICLE

A Thesis

by

CRAIG ALLEN ODOM

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Make McDermott
Committee Members,	Glen Williams
	Darbha Swaroop
Head of Department,	Dennis O'Neal

August 2006

Major Subject: Mechanical Engineering

ABSTRACT

Navigation Solution for the Texas A&M Autonomous Ground Vehicle. (August 2006)

Craig Allen Odom, B.S., Texas A&M University

Chair of Advisory Committee: Dr. Make McDermott

The need addressed in this thesis is to provide an Autonomous Ground Vehicle (AGV) with accurate information regarding its position, velocity, and orientation. The system chosen to meet these needs incorporates (1) a differential Global Positioning System, (2) an Inertial Measurement Unit consisting of accelerometers and angular-rate sensors, and (3) a Kalman Filter (KF) to fuse the sensor data. The obstacle avoidance software requires position and orientation to build a global map of obstacles based on the returns of a scanning laser rangefinder. The path control software requires position and velocity.

The development of the KF is the major contribution of this thesis. This technology can either be purchased or developed, and, for educational and financial reasons, it was decided to develop instead of purchasing the KF software. This thesis analyzes three different cases of navigation: one-dimensional, two dimensional and three-dimensional (general). Each becomes more complex, and separating them allows a three step progression to reach the general motion solution.

Three tests were conducted at the Texas A&M University Riverside campus that demonstrated the accuracy of the solution. Starting from a designated origin, the AGV traveled along the runway and then returned to the same origin within 11 cm along the North axis, 19 cm along the East axis and 8 cm along the Down axis. Also, the vehicle traveled along runway 35R which runs North-South within 0.1°, with the yaw solution consistently within 1° of North or South. The final test was mapping a box onto the origin of the global map, which requires accurate linear and angular position estimates and a correct mapping transformation.

ACKNOWLEDGMENTS

I would like to thank my committee chair, Dr. McDermott, and my committee members, Dr. Williams and Dr. Swaroop, for their assistance and guidance during my research. I am very grateful for the wonderful project that I was and continue to be part of throughout my graduate career. I have gained an abundance of knowledge inside and outside of the classroom.

Thanks also to my family and friends who have supported me throughout my college career. I have had my high and low moments, but the love and support of my family and friends have remained steadfast and high.

I would like to especially acknowledge the patience and love of my wife.

NOMENCLATURE

A_{pitch}	The rotation matrix for Pitch
A_{roll}	The rotation matrix for Roll
A_{sweep}	The rotation matrix for Sweep
A_{yaw}	The rotation matrix for Yaw
ARS	Angular-rate sensor
$C(\text{diag}(i-j))$	Denotes the diagonal from row i to row j of matrix C
DCM_{ICS}^{BCS}	Matrix that maps a vector with components in the Inertial Co-ordinate System (ICS) to components in the Body Co-ordinate System (BCS)
$E\{x\}$	Expected value
$f_i(t)$	i-th function
$\mathrm{f}(\mathrm{x}(t),\mathrm{u}(t),t)$	Vector of nonlinear functions defining the derivatives of the state vector
$F(\hat{\mathrm{x}}(t),t)$	Partial derivatives of $\mathrm{f}(\mathrm{x}(t),\mathrm{u}(t),t)$ used for propagation
g	Gravitational constant at the surface of the Earth (~ 9.807 m/s/s)
$G(t)$	Matrix that maps the process noise into the nonlinear functions
$\mathrm{h}(\mathrm{x}_k)$	Vector of GPS position measurements as functions of the states
$\mathrm{h}(\hat{\mathrm{x}}_k^-)$	Vector of predicted GPS measurements
$H_k(\hat{\mathrm{x}}_k^-)$	Measurement sensitivity matrix utilizing the *a priori* state estimate at discrete-time t_k
K_k	Kalman gain matrix at discrete-time t_k
m	Range value, in meters, from the origin of the SICK co-ordinate system to the SICK hit, with components in x_{SICK} and y_{SICK} axes (with β as the angle provided by the SICK)
$N..E..D$	The primary axes of the ICS (North, East and Down)
$N'..E'..D'$	The secondary axes of the ICS (after the Yaw rotation)
$N''..E''..D''$	The third set of axes of the ICS (after the Pitch rotation)

$N(0,\ Q(t))$	Normal distribution with zero mean and covariance matrix $Q(t)$
$N(0,\ R_k)$	Normal distribution with zero mean and covariance matrix R_k
$p,\ q,\ r$	Body-fixed angular rates about the x, y and z axes, respectively
$\tilde{p},\ \tilde{q},\ \tilde{r}$	ARS measurements (in °/s) about the x, y and z axes, respectively
P_0	Initial estimation error covariance matrix
$P(t)$	Continuous-time estimation error covariance matrix
$\dot{P}(t)$	Continuous-time, time derivative of estimation error covariance matrix
P_k^-	*a priori* estimation error covariance matrix at discrete-time k
P_k^+	*a posteriori* estimation error covariance matrix at discrete-time k
$Q(t)$	Process noise error covariance matrix
$\vec{r}_{h/s,s}$	Vector that locates the SICK hit with respect to the origin of the SICK co-ordinate system, in SICK co-ordinates
$\vec{r}_{s/i,i}$	Vector of offsets of SICK origin from IMU origin in IMU co-ordinate system
R_k	Measurement noise error covariance matrix
SFT	Scale factor transformation
t	Continuous-time
$\mathrm{u}(t)$	Vector of true inputs
$v^2_{GPS,N..E..D}$	Variance of the GPS position measurement error along the N, E and D axes, respectively
v_k	GPS position measurement noise at discrete-time k
$w^2_{Acc,x..y..z}$	Variance of the accelerometer measurement error along the x, y and z axes, respectively
$\mathrm{w}(t)$	Continuous-time process noise
$x..y..z$	The axes of the SICK or IMU BCS
$x_i(t)$	i-th state
$\mathrm{x}(t)$	Vector of true states
$\dot{\mathrm{x}}(t)$	Vector of time derivatives of true states
$\hat{\mathrm{x}}(t_0)$	Initial state estimate vector

$\hat{\mathrm{x}}(t)$	Continuous-time vector of estimated states used for propagation
$\dot{\hat{\mathrm{x}}}(t)$	Continuous-time vector of time derivatives of estimated states
x_k	True state at discrete-time k
$\hat{\mathrm{x}}_k^-$	*a priori* state estimate at discrete-time k
$\hat{\mathrm{x}}_k^+$	*a posteriori* state estimation vector at discrete-time k
$\ddot{\tilde{x}}_k$, $\ddot{\tilde{y}}_k$, $\ddot{\tilde{z}}_k$	Accelerometer measurements (in m/s/s) along the x, y and z axes, respectively
$\tilde{\mathrm{y}}_k$	GPS position measurements available at discrete-time k
β	Angle to object in SICK co-ordinate system that is provided by SICK
$\delta x_{s/i}$	Offset of SICK origin from IMU origin along x_{IMU} axis (constant)
$\delta y_{s/i}$	Offset of SICK origin from IMU origin along y_{IMU} axis (constant)
$\delta z_{s/i}$	Offset of SICK origin from IMU origin along z_{IMU} axis (constant)
η_b	Determines the level to which the bias will randomly walk (0 if no walk)
θ	Pitch angle defined as the amount of rotation about the E' axis from the N' axis to the x_{IMU} axis
$\dot{\theta}$	Pitch rate
$\dot{\mu}$	Time derivative of a bias (either accelerometer or ARS)
$\dot{\hat{\mu}}$	Estimated time propagation of the bias (KF has no knowledge of dynamics of bias, therefore it is 0)
ϕ	Roll angle defined as the amount of rotation about the N'' axis from the E'' axis to the y_{IMU} axis
$\dot{\phi}$	Roll rate
ψ	Yaw angle defined as the amount of rotation about the D axis from the N axis to the projection of the x_{IMU} axis onto the $N-E$ plane
$\dot{\psi}$	Yaw rate
$\dot{\zeta}$	Time derivative of ζ (used as an example; ζ not located in text)

$\ddot{\zeta}$	Second time derivative of ζ
$\tilde{\zeta}$	Value of measurement of ζ
$\hat{\zeta}$	Estimated value of ζ

TABLE OF CONTENTS

Page

Page

LIST OF FIGURES

Page

Page

Page

LIST OF TABLES

Page

1. INTRODUCTION

Navigation is essential to the operation of an Autonomous Ground Vehicle (AGV). The current position, velocity and yaw of the AGV are required to control the vehicle's movements through a series of waypoints. A Global Positioning System (GPS) receiver only receives position information and then computes the velocity and heading (not yaw) of the vehicle. Heading is defined as the direction of travel, which can be computed by using the North and East velocities. The yaw angle describes where the front of the vehicle is pointing rather than where it is going. For example, if the vehicle were traveling in reverse, heading and yaw would be 180° from each other. Yaw requires an angular rotation to change, whereas heading requires a change in the North and/or East velocities.

GPS cannot be the sole provider of navigation information because it is susceptible to outages and jamming and cannot provide orientation information. During GPS outages, the AGV has no state information and thus is unable to make any decisions concerning a new direction of travel. Losing all navigation information is an unacceptable situation, especially during deployment in a military setting. However, the AGV does not house any military personnel, making it an alternative to sending in manned vehicles.

In order to reduce the effect of outages and jamming, the GPS information is fused with information from an Inertial Measurement Unit (IMU) that has three orthogonal accelerometers and three orthogonal angular-rate sensors (ARS's). The GPS position information has a large error associated with it, but it is bounded. The accelerations and angular rates from the IMU must be integrated which introduces error that can grow unbounded. Each of these sensors are useful separately but when combined they are exceptional.

Before purchasing these sensors, a design analysis was required to develop a needs statement and functional/performance requirements. This is a very systematic, top-down approach to purchasing sensors. The benefit is that the sensors purchased should meet the requirements of the project, reducing some of the problems associated with integrating new sensors.

This thesis follows the style of Journal of Dynamic Systems, Measurement, and Control.

The outputs from these two sensors are combined by the Kalman Filter (KF) to produce a "best" estimate of the actual AGV states. With knowledge of the accuracy of the sensors and a kinematic model the KF develops a weighting matrix. This weight matrix quantifies how much the model will be corrected by the measurements.

This thesis separates the analysis of the navigation system into four parts. The first three sections describe the simulations associated with particular motions that the AGV will undergo. The first is one-dimensional (1-D) motion, the second is two-dimensional (2-D) motion and the third is three-dimensional (3-D) or general motion. The last section of the analysis shows the results of using the general motion KF on the real AGV.

The 1-D model is the simplest motion possible. The simulated AGV motion is driving North with no angular rotations. There is one GPS position measurement (latitude) and one accelerometer measurement available. The purpose of simulating 1-D motion is to create the foundation for the other two motions while maintaining the simplest configuration possible. Debugging the 1-D model is also much simpler than attempting to debug the 3-D model outright.

The 2-D model incorporates two GPS position measurements (latitude and longitude), two accelerometer measurements and one ARS measurement. This model is valid if the AGV travels on a flat plane, normal to the gravity vector of the Earth (thereby removing any gravity components in the accelerations). There is no pitching or rolling in this model, but the AGV will pitch and roll during braking/acceleration and turning. This motion will affect the accelerometer outputs by adding the gravitational component due to the pitch and roll angles. Performance will degrade as the pitch and roll angles increase, but the effect is quite small at lower angles ($< 5°$).

The 3-D model is the general model. It incorporates three GPS position measurements (latitude, longitude and height above sea level), three accelerometer measurements, and three ARS measurements. This model is valid for all motion, except if the AGV pitches $\pm 90°$, wherein there is a singularity. During normal operation, the AGV will not experience a pitch angle of $\pm 90°$.

It should be noted that the latitudes, longitudes and heights above sea level provided by the GPS receiver are converted into displacements from a designated origin. For the simulations, the displacements are generated rather than creating latitudes and longitudes and then converting them to displacements. Implementing the general motion KF for use on the AGV requires the conversion of GPS measurement data to displacements along the North, East and Down axes. Refer to section four regarding details of this transformation.

The KF also provides state information to the obstacle avoidance software. The information used by the software is: North, East and Down position of the AGV, the Euler angles (yaw, pitch and roll), offsets of the SICK from the IMU and the SICK returns (range and angle). Once the software has this information it can build an accurate global map of the surrounding environment. Additional work was required to cluster the hits to create obstacles that the AGV must avoid.

2. CHOICE OF SENSORS

A design analysis is a systematic method for choosing an option, whether it is a product to meet some demand, or a choice concerning an action. For the purposes of this project, it was used to purchase sensors for the AGV. Initially, the desire was to purchase an Integrated Navigation System (INS) which would provide the AGV with all the required states at 100 Hz. The INS includes a GPS receiver, an IMU and a KF. However, due to cost restrictions, the entire system was not purchased. Rather, only the GPS receiver was purchased, with intentions of purchasing the IMU separately at a later time along with the integration software support, which contains the KF.

Three items are necessary in a design analysis: Needs statement, functional/performance requirements, and a list of options that could resolve the needs statement. The needs statement is the most important step in a design analysis. This is the initial step from which the other steps must follow. Posing the needs statement correctly and capturing exactly what the sensor's purpose is critical. The needs statement for the INS is the following:

"Utilize integrated GPS, IMU and a Kalman filter to provide fast and accurate linear and angular position/velocity states for an AGV at a cost compatible with project budget".

This is not the most general needs statement that could be posed. However, it was decided very early in the project that a GPS/IMU/KF combination would be used for navigation and mapping. This combination was used extensively by the teams competing in the DARPA Grand Challenge [1], thus it was a good starting point. Other means of providing vehicle states were discussed, but their usefulness for this project was limited.

The needs statement is the basis for the functional/performance requirements. The functional requirements are the functions the INS must perform, i.e. providing AGV states fast enough to raise the maximum stable vehicle speed. The performance requirements are how well the functional requirements are performed, i.e. providing state information to the AGV at six Hz. The performance requirements must be able to be tested because they are quantitative in nature. The functional/performance requirements are listed below, with a short justification for each performance requirement.

FR # 1: Provide position and velocity readings to path controller fast enough to increase maximum stable vehicle speed to 40+ mph.

PR # 1: Navigation system must update states at Six Hz.

Justification: *The digital path controller is very reliant on the update rate. As time delays in state feedback increase, controller performance will degrade and eventually become unstable. The higher the update rate, the better the controller will perform (assuming good controller design), within AGV response limits.. Previously, the update rate was one Hz, with a maximum stable vehicle speed ~ 15 mph. If a linear trend is assumed, a three Hz update rate would be required to achieve a maximum stable vehicle speed of 40+ mph. Therefore, a six Hz update rate is specified which includes a safety factor of two.*

FR # 2: Provide accurate linear/angular position and velocity states to AGV.

PR # 2: Linear position error less than $1/3$ meter. Yaw angular position error less than $0.5°$.

Justification: *The path width on the DARPA course can be narrow at times (three meters). With the AGV track width of two meters, this leaves* $1/2$ *meter on either side, thus requiring a position error less than* $1/3$ *meter. Also, the obstacle avoidance requires accurate attitude information to build the obstacle map. At 60 meters, with an error of 0.5° in the yaw angle, a SICK hit will have a global position error of* $1/2$ *meter, which is an acceptable error. The yaw angle is the least accurate of the three Euler angles and thus it is used as the requirement.*

FR # 3: IMU must have high operating range.

PR # 3: Linear operating range greater than 10g. Angular operating range greater than 200°/s.

Justification: *The required operating range is dictated by the AGV and operating conditions. It is not advisable to saturate the operating range of the IMU. The maximum rotational speed and heave acceleration experienced by the AGV were approximated by testing and analysis to be 100°/s and five g's, respectively. With a safety factor of two, the required operating ranges are 200°/s and 10g's.*

FR # 4: Navigation system must include a KF.

PR # 4: KF must be capable of fusing IMU/GPS data.

Justification: *A KF is required to provide orientation information (not provided by GPS or IMU) and continuosly provide all vehicle states, including during GPS outages.*

FR # 5: IMU must have low drift rate.

PR # 5: Drift rate needs to be less than 30°/hr.

Justification: *The biases for the three accelerometers and three ARS's will drift. The KF continually compensates for these biases, but it requires that a GPS signal is available. Without GPS, the error in the states will grow without bound. Assuming that a one minute loss in GPS is a reasonable amount of time, and a yaw angle error of 0.5° is an acceptable error, then a 30°/hr drift is a reasonable requirement.*

FR # 6: GPS receiver must be able to acquire a commercially available Differential GPS (DGPS) signal, i.e. OmniSTAR.

PR # 6: Must be able to acquire DGPS signal in Mojave Desert.

Justification: *This is a two-fold requirement. DGPS provides sub-meter position accuracy, which is required by the path controller. The other is that DARPA requires that a commercially or publicly available differential signal be used (no proprietary differential correction). OmniSTAR is an example of a reliable, satellite-based differential signal with wide geographic availability.*

FR # 7: IMU/GPS communication must be based on protocol familiar to AGV team.

PR # 7: Communication must be serial or TCP/IP.

Justification: *The AGV team is most familiar with these protocols, reducing the development time.*

FR # 8: INS must be robust in shock and vibration.

PR # 8: Shock survivability must be greater than 20 g for 5 ms.

Justification: *During testing, the AGV experienced a maximum of five-g heave acceleration for five ms while driving over large parking lot dividers at 20mph. Therefore, the shock survival was chosen to be 20 g for five ms to give an additional factor of safety. It is imperative that the INS survive these accelerations.*

A search was performed to identify candidate INS systems. The list of candidates is given in table 1. The NovaTel INS [2] with the G2-H58 IMU (SPAN # 2) is the best candidate because it meets all the requirements (except shock robustness) and is less expensive than the SPAN # 1 package. Once funds become available, the SPAN # 2 package will be the final solution, outside of purchasing the support software.

Table 1. Candidate list with functional/performance requirements

Func. Req.	Update Rate	Linear / Angular Accuracy	Linear / Angular Operating Range	Kalman Filter?	Drift Rate	DGPS comm. Signal?	Com.	Shock Robust	Price / Lead Time
Perf. Req.	6 Hz	1/3 m / 0.5°	10 g / 200°/s	Yes	30°/hr	Yes	Serial or TCP/IP	20 g for 5 ms	Minimal
Applanix POS LV	1 - 200 Hz	Not provided / 0.07°	Not provided	Not provided	Not provided	Not provided	RS232	Not provided	$65,000 / 4 - 8 wks
Crossbow NAV 420	2 - 100 Hz	3 m / 0.5°	4 g / 200°/s	Yes	2700°/hr	No	RS232	200 g for 1 ms	$9,500 / 3 - 4 wks
NovaTel SPAN # 1	100 Hz	0.10 m / 0.05°	50 g / 1000°/s	Yes	1°/hr	Yes	RS232 or RS422	100 g for 14 ms	$23,000 / 4 wks
NovaTel SPAN # 2	100 Hz	0.10 m / 0.05°	50 g / 1000°/s	Yes	10°/hr	Yes	RS232 or RS422	100 g for 14 ms	$13,000 / 4 wks
OTS RT3040	100 Hz	0.10 m / 0.03°	30 g / 300°/s	Yes	Not provided	Yes	Ethernet UDP	100 g for 11 ms	$52,000 / 4 - 8 wks
Micro - botics MIDG II	50 Hz	2 m / 0.1°	10 g / 300°/s	Yes	150°/hr	Yes	Serial	100 g for 8 ms	$5,800 / No lead

3. KALMAN FILTER AND SENSOR INTEGRATION

Measurements from sensors are inherently noisy and have biases that fluctuate with time. When taking a measurement, the engineer or scientist wants the truth that is masked by the biases and noise. The truth is never known outside of a simulation where it is generated by the software. Estimating the truth requires some knowledge of the noise in the sensor and the physics of the action that the sensor is monitoring. The estimator used most often for accurate state information of dynamic systems is the Kalman filter.

The KF is an optimal estimator that minimizes the variance between a propagated state (model) and an observation (measurement). Neither the model nor the measurement is perfect, thus requiring that the information from both is fused together at a ratio determined by the KF. The fused information is more accurate than either the model or the measurement, even if one is much better than the other [3].

The dynamics of the model are based on either equations of motion developed by the engineer, or a kinematic model. A kinematic model is one based solely on positions, velocities and accelerations and does not include or require knowledge of forces or moments. This is the model to use when accelerometers/ARS's are available.

The main assumption inherent in the KF is that the noise is white with zero mean. At any instant, the value of the noise does not depend on any prior values, and it follows the normal distribution curve (bell curve) developed by Gauss. This holds for many instances, including IMU and GPS dynamics, and is therefore a reasonable assumption.

The KF estimates a state that is governed by a model. The general form of the truth model is given in equation (3.1) [4].

$$\dot{\mathrm{x}}(t) = \mathrm{f}(\mathrm{x}(t), \mathrm{u}(t), t) + G(t)\mathrm{w}(t) \tag{3.1}$$

The notation conventions for the variables in the equations provide insight into what the equation is to accomplish. The list below provides the notation conventions that are followed throughout this thesis.

- A bold variable represents a vector, e.g., $\mathbf{x}$
- A lowercase variable that is not bold represents a scalar, e.g., t
- An uppercase variable represents a matrix, e.g., G
- A variable with a ^ represents an estimate, e.g., $\hat{\mathbf{x}}$
- A variable with a ~ represents a measurement, e.g., $\tilde{\mathbf{y}}$
- A variable with nothing above it represents the truth, e.g., $\mathbf{x}$
- The subscript (e.g., k) represents a value at a discrete time t_k
- A variable with a superscript, $-$, represents the *a priori* estimate, e.g., $\hat{\mathbf{x}}_k^-$
- A variable with a superscript, $+$, represents the *a posteriori* estimate, e.g., $\hat{\mathbf{x}}_k^+$

In equation (3.1), the propagation of the state vector is dependent on the state itself $\mathbf{x}(t)$, the input $\mathbf{u}(t)$, time t and the process noise $\mathbf{w}(t)$. The process noise is the error in the model from the true process. There is also a measurement model used by the KF [4]:

$$\tilde{\mathbf{y}}_k = \mathbf{h}(\mathbf{x}_k) + \mathbf{v}_k \tag{3.2}$$

Where: $\tilde{\mathbf{y}}_k$ is the vector of GPS position measurements given to the KF

$\mathbf{v}_k$ is the measurement noise

The states and the measurements are not always of the same parameter. For instance, if the states were North and East positions, but the measurement was heading, there is not a 1:1 relationship. The vector of functions $\mathbf{h}(\mathbf{x}_k)$ maps the states into the GPS position measurements.

The two models each have noise. The variables $\mathbf{w}(t)$ and $\mathbf{v}_k$ represent the process noise and measurement noise, respectively. Each are white Gaussian noise with zero mean. This can be stated as [4]:

$$\begin{aligned} \mathbf{w}(t) &\sim N(0, Q(t)) \\ \mathbf{v}_k &\sim N(0, R_k) \end{aligned} \tag{3.3}$$

$Q(t)$ is the process noise covariance matrix, and R_k is the measurement noise covariance matrix. These matrices arise from the expected value of their respective noises, namely [4]:

$$E\left\{\mathrm{w}(t)\mathrm{w}(t)^T\right\} = Q(t)$$
$$E\left\{\mathrm{v}_k\mathrm{v}_k^T\right\} = R_k \tag{3.4}$$

Once the models have been developed, which is indeed the most time-consuming and difficult step, the KF algorithm can be implemented. The KF needs the initial state estimates $\hat{\mathrm{x}}(t_0)$ as well as the uncertainty in these initial state estimates P_0. The P_0 matrix is the initial estimation error covariance matrix, or the initial information matrix. This matrix is pivotal to the operation of the KF and is used to determine the Kalman gain matrix.

Yielding good performance from the KF is highly dependent on P_0. If the values chosen for the diagonals of P_0 are set too low (little uncertainty about the initial estimates) the KF estimates will be sluggish and not track the dynamics of the system very well. If they are set too high the KF estimates will be very sensitive to the noise in the measurements and will also not track very well.

There is not a unique solution for P_0. The nonlinear functions in equation (3.8) and cross-coupling of the states makes it very difficult to provide a closed-form solution for P_0. There are formulae for determining the P_0 matrix, but mostly P_0 is chosen by trial and error on a system simulator. A good approach is to find a state that initially is known very well and have its diagonal element set to one. Then increase the value of the diagonal elements of P_0 based on the corresponding uncertainty of their initial estimates. For example, the scale factor transformation will be very close to one (which is indeed the initial estimate) and thus that diagonal element is set to one. However, the orientation of the vehicle is not known very well and thus the diagonal elements corresponding to the yaw, pitch and roll angles are set to 10. This process is done for all the states. There is much additional work required to fine tune the P_0 matrix to provide the performance that is desired.

Once $\hat{\mathrm{x}}(t_0)$ and P_0 are chosen, the KF algorithm can begin with the first GPS position measurement that is available. During the periods between these measurements, the KF is not

functioning; rather the model is used without correction. An uncorrected solution will drift with time, thus requiring a sufficiently fast update rate from the GPS.

Once the first set of GPS position measurements is available, the Kalman gain matrix K_k can be determined [4] using the gain propagation equation (3.5). This equation is very important to the function of the KF algorithm.

$$K_k = P_k^- H_k^T(\hat{\mathrm{x}}_k^-)\left[H_k(\hat{\mathrm{x}}_k^-) P_k^- H_k^T(\hat{\mathrm{x}}_k^-) + R_k \right]^{-1}$$
$$H_k(\hat{\mathrm{x}}_k^-) \equiv \left.\frac{\partial \mathrm{h}}{\partial \mathrm{x}}\right|_{\hat{\mathrm{x}}_k^-} \tag{3.5}$$

Initially, the P_0 matrix becomes P_k^- and $\hat{\mathrm{x}}(t_0)$ becomes $\hat{\mathrm{x}}_k^-$. Once the Kalman gain matrix has been determined, it is used to correct the state estimation vector. Beyond the initial period, the state estimation vector will be propagated by a model (shown later) which produces the *a priori* estimate. This *a priori* estimate is corrected by the Kalman gain matrix acting on the difference between the predicted measurement and the actual measurement. The new estimate is the *a posteriori* estimate which is given to the AGV because it is the best available estimate. The equation below demonstrates this process [4].

$$\hat{\mathrm{x}}_k^+ = \hat{\mathrm{x}}_k^- + K_k\left[\tilde{\mathrm{y}}_k - \mathrm{h}(\hat{\mathrm{x}}_k^-) \right] \tag{3.6}$$

The information matrix is corrected by the following equation [4].

$$P_k^+ = \left[I - K_k H_k(\hat{\mathrm{x}}_k^-) \right] P_k^- \tag{3.7}$$

Equation (3.7) demonstrates that the correction from the KF reduces the uncertainty of the estimation. The P_k^+ matrix is referred to as the information matrix because it contains information vital to the KF. The elements of the information matrix should be decreasing as time progresses; otherwise the solution will be unstable.

The final step in the KF algorithm is the propagation. The *a posteriori* estimates are used as the initial conditions to propagate to the next time step via an integration tool (see equation 3.8).

$$\begin{aligned} \dot{\hat{\mathrm{x}}}(t) &= \mathrm{f}(\hat{\mathrm{x}}(t), \mathrm{u}(t), t) \\ \dot{P}(t) &= F(\hat{\mathrm{x}}(t), t)P(t) + P(t)F^T(\hat{\mathrm{x}}(t), t) + G(t)Q(t)G^T(t) \\ F(\hat{\mathrm{x}}(t), t) &\equiv \left.\frac{\partial \mathrm{f}}{\partial \mathrm{x}}\right|_{\hat{\mathrm{x}}(t)} \end{aligned} \tag{3.8}$$

The state estimates are propagated by the process model. The information matrix is propagated by a form similar to the Riccati equation [4]. Recalling that $Q(t)$ is the process noise covariance matrix, it can be seen in the information matrix propagation equation that the process noise will try to increase the uncertainty in the estimation. The update in equation (3.7) reduces uncertainty in the estimates, whereas the propagation in equation (3.8) increases the uncertainty. Therefore, it is desired to have the fastest measurement update rate possible to reduce the effect $Q(t)$ has on the certainty of the estimates.

The results of the propagation step are the *a priori* estimates, $\hat{\mathrm{x}}_{k+1}^{-}$ and P_{k+1}^{-}. When a measurement is not available, there are no updates and thus the propagation continues. Between measurements, the uncertainty in the estimates grows because the P matrix is growing.

4. SIMULATION

The KF code was tested first in simulation and then in the AGV with the GPS and IMU. The general motion KF was the only model to be tested in the AGV. The simulation tests are efficient in terms of time and resources. Also, any problems that arise in the simulation can be fixed and do not endanger life or property. Thus, it is important to mimic the environment that the AGV will encounter. With the simulation, the truth is available and can be used to evaluate and tune the KF. The simulation implements the process and measurement models required by the KF and adds the noise, bias and scale factor transformation errors (explained later).

The displacements along the North, East and Down axes from the origin are defined as the position of the AGV. However, GPS provides latitude, longitude and height above sea level. These need to be converted to displacements for use in the KF. The transformation used is the following:

$$\begin{aligned} N &= 2R_{Earth}\sin\left(\frac{Lat_c - Lat_o}{2}\right) \\ E &= 2R_{Earth}\sin\left(\frac{Lon_c - Lon_o}{2}\right)\cos\left(\frac{Lat_c - Lat_o}{2}\right) \\ D &= H_o - H_c \end{aligned} \tag{4.1}$$

Where: $R_{Earth} = 6366564.864m$

Lat_c, Lon_c, H_c are the current latitude, longitude and height above sea level of the AGV, respectively

Lat_o, Lon_o, H_o are the latitude, longitude and height above sea level of the origin of the Inertial Co-ordinate System, respectively (explained later)

Starting with a simple example makes it much easier to debug both the simulation and the KF. For this reason, the initial case that was simulated is 1-D motion. Only one GPS position measurement (North) and one accelerometer measurement are available. This case models the vehicle traveling in a straight line with no rotations. Each measurement is corrupted with white, Gaussian noise with zero mean (the ideal noise characteristics for the KF). Also, there is a bias applied to the accelerometer reading that, if uncompensated, will reduce the accuracy of the estimated positions and velocities very quickly. This is known as integration error. Also, the bias is simulated as having a random walk to make its dynamics more realistic.

A random walk is defined as the current value of the noise being dependent on the previous value. It is unstable, but can be compensated for by the KF. The model for a bias with random walk is the following [4]:

$$\dot{\mu} = \eta_b \tag{4.2}$$

Where: $\dot{\mu}$ is the time-derivative of the bias and η_b is a random number

If η_b were zero, no random walk would exist. The model used in the KF is the following [4]:

$$\dot{\hat{\mu}} = 0 \tag{4.3}$$

This states that the KF assumes that the bias is constant thus the KF has no information about the bias dynamics. However, the value of η_b is usually very small, on the order of e-7 for a good sensor, and thus the random walk of the bias is low. The value of η_b is not known for the Watson Industries IMU. The slower the random walk of the bias, the easier the KF can compensate for it.

The other corruption estimated by the KF is the scale factor transformation (SFT). When the accelerometer senses acceleration, the sensor outputs a voltage. This is a V/g scaling. Then, the microcontroller inside the IMU applies the reverse scaling to provide a 13 bit number (plus a sign bit) proportional to the sensed acceleration to the communication port. Thus, the SFT should be one in this configuration. However, the sensor does not always apply the correct scaling to the sensed acceleration. Estimating this variable instead of assuming it is one provides greater flexibility for the KF. Since the value should be close to one this is the initial estimate.

Once the 1-D case is solved and the results are satisfactory, the next step is to simulate the 2-D case. For this, there are two GPS position measurements (North and East), two accelerometer measurements (x, y) and one ARS measurement. This case models an AGV traveling on a flat plane, normal to the gravity vector of the Earth, while maintaining a perfectly upright orientation (no pitch or roll). The ARS measurement is necessary because the accelerometers are measuring accelerations in a *body-fixed* co-ordinate system, which is not an *inertial* co-ordinate system. The angular rate measurement provides information about the orientation of the vehicle (body) with respect to the inertial co-ordinate system.

The Inertial Co-ordinate System (ICS) is a plane normal to the gravity vector on the earth's surface, having axes North, East and Down. The origin of the ICS is a designated waypoint. North, East and Down (in that order) define a right-hand co-ordinate system. The Body-fixed Co-ordinate System (BCS) is based on SAE co-ordinates, having axes x, y and z, where +x is toward the front of the vehicle, +y is toward the passenger side and +z is down. The origin of the BCS is the location of the IMU.

The final case is the 3-D, or general case. This includes three GPS position measurements (North, East and Down), three accelerometer measurements (x, y and z) and three ARS measurements (p, q and r). This is the general (and most difficult) case. It can be reduced to the 1-D or 2-D cases if appropriate assumptions are made. The analysis will start with the 1-D case and finish with the 3-D case.

4.1 1-D Model

The 1-D motion utilizes one accelerometer measurement and the GPS North position measurement. The accelerometer measurement contains a bias that walks over time and a SFT error. The GPS measurements were simulated as having a one-sigma error of one meter with zero mean. There are no biases or SFT's for the GPS measurements because they are not observable by the KF. Differential GPS removes most of the bias in the position measurement. The IMU characteristics used for the simulation were based on the Watson Industries IMU-BA604 [5] because it is the IMU currently available (located in table 3). The final solution IMU will perform much better, but the additional development of the KF dynamics will change very little.

The 1-D motion has the characteristics listed in table 2 and equations (4.4) through (4.8). These are the equations used in the simulation and are vital to the operation of the KF.

Table 2. List of states/parameters for 1-D case

$x_1(t)$	North position
$x_2(t)$	North velocity
$x_3(t)$	Bias in x-axis accelerometer
$x_4(t)$	SFT for x-axis accelerometer

Model

$$\begin{aligned} \dot{x}_1(t) &= x_2(t) \\ \dot{x}_2(t) &= x_4(t)\left(\ddot{\tilde{x}}_k - x_3(t)\right) \\ \dot{x}_3(t) &= 0 \\ \dot{x}_4(t) &= 0 \end{aligned} \tag{4.4}$$

Initial State Estimate Vector

$$\hat{\mathrm{x}}(t_0) = \begin{bmatrix} 0 & 25 & 0 & 1 \end{bmatrix}^T \tag{4.5}$$

Measurement and Process Noise Covariance Matrices

$$R_k = v_{GPS,N}^2 = 1m^2,\ Q(t) = w_{Acc,x}^2 = (0.005g)^2 \tag{4.6}$$

Process Noise Mapping Matrix

$$G(t) = \begin{bmatrix} 0 & x_4(t) & 0 & 0 \end{bmatrix}^T \tag{4.7}$$

Initial Estimation Error Covariance Matrix

$$P_0 = \mathrm{diag}\begin{bmatrix} 0.5 & 0.5 & 0.005 & 0.005 \end{bmatrix} \tag{4.8}$$

The simulation was run for 200 seconds. Multiple maneuvers were tested including constant acceleration, random acceleration and sinusoidal acceleration. The results for the sinusoidal acceleration are shown in this thesis. The path of the vehicle along the North axis can be seen in figure 1 and is shown in equation (4.9).

$$N(t) = 200\sin\left(2\pi\frac{t}{50}\right) \tag{4.9}$$

Figures 2-6 show that the KF requires some time to converge while it is getting educated about the dynamics of the system. After approximately 60 seconds the bias and SFT have converged and the filtered outputs are very close to the truth. Again, the truth is not known in the real world, but the simulation knows the truth and thus the error can be calculated. Figure 2 displays the North position estimation error. Figure 3 displays the error of the North velocity estimate. Figure 4 displays the error of the x-axis accelerometer bias estimate. Figure 5 displays the SFT percentage error of x-axis accelerometer estimate. Figure 6 displays the random walk of the x-axis accelerometer.

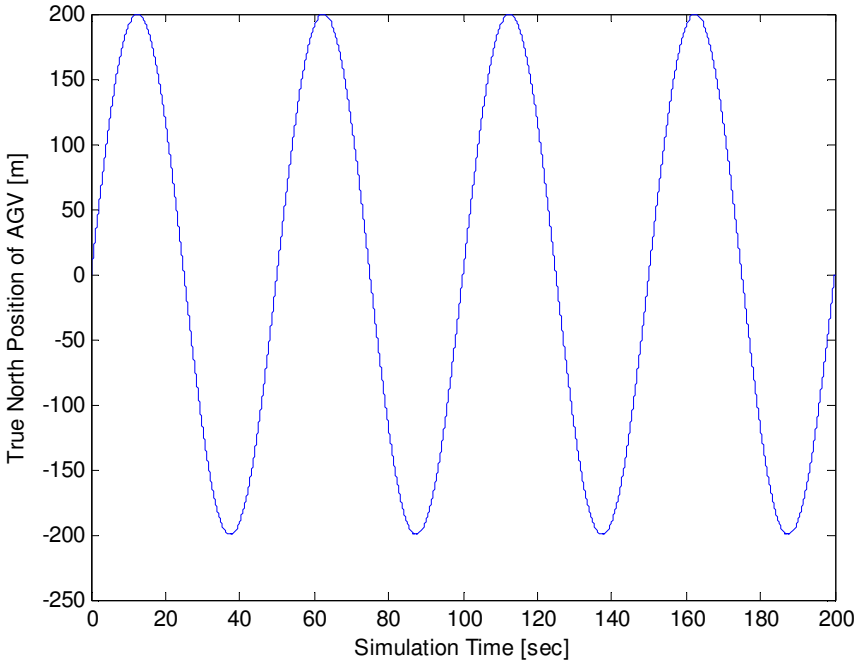

Figure 1. True North position of AGV for 1-D case

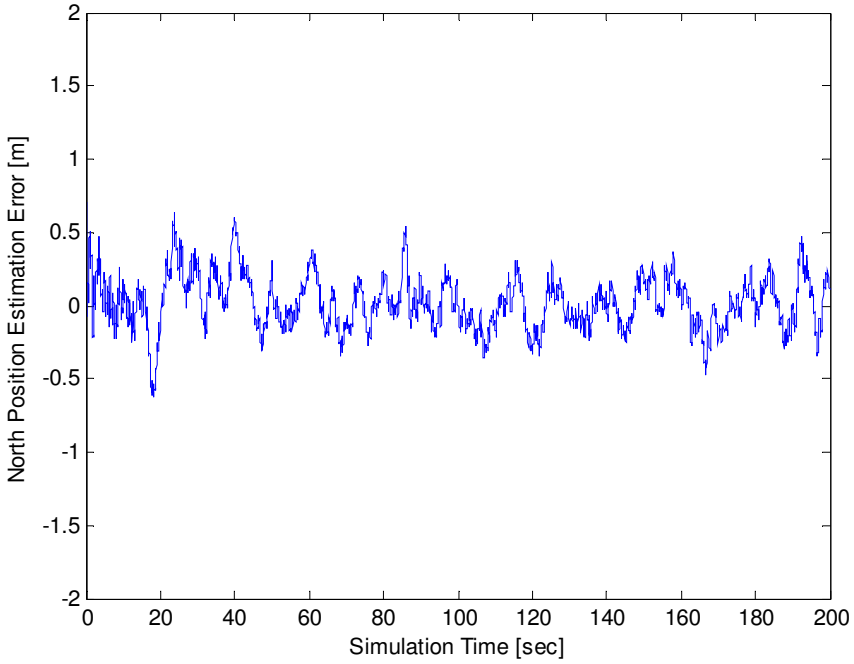

Figure 2. North position estimation error for 1-D case

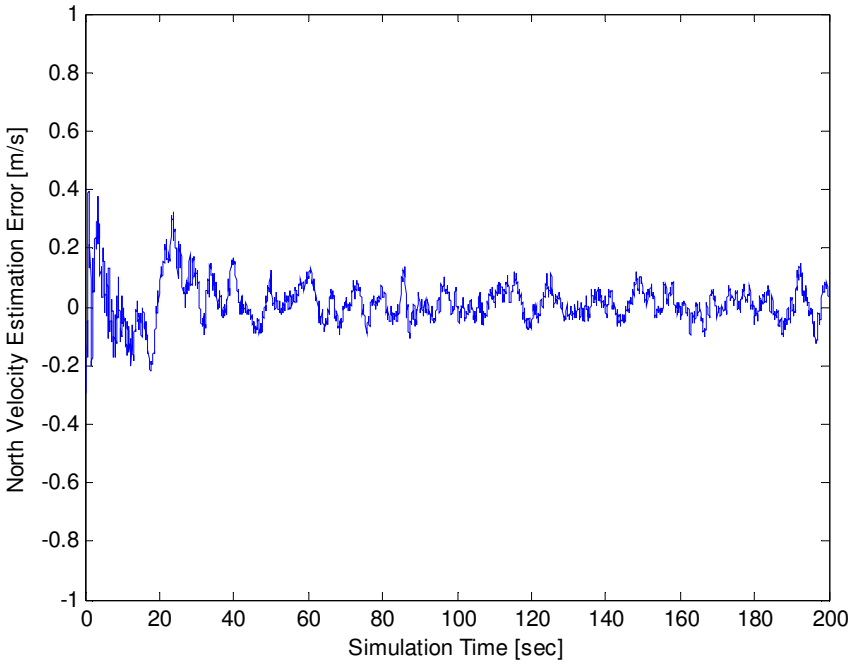

Figure 3. North velocity estimation error for 1-D case

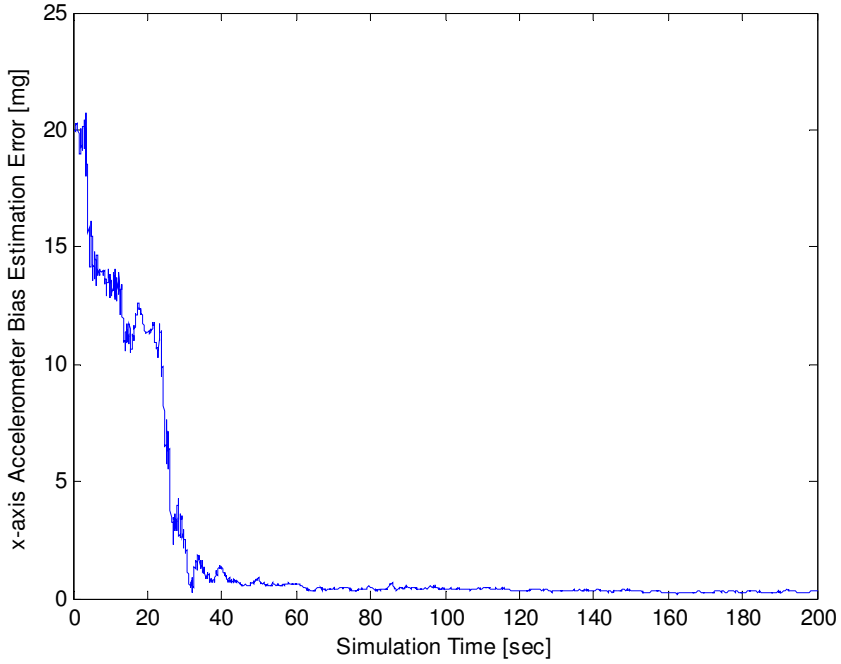

Figure 4. x-axis accelerometer bias estimation error for 1-D case

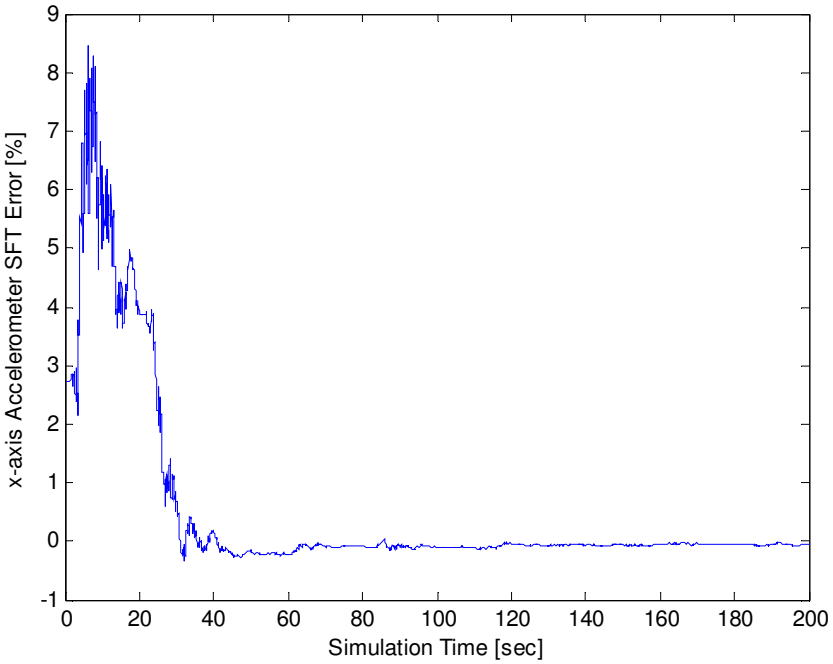

Figure 5. x-axis accelerometer SFT percentage error for 1-D case

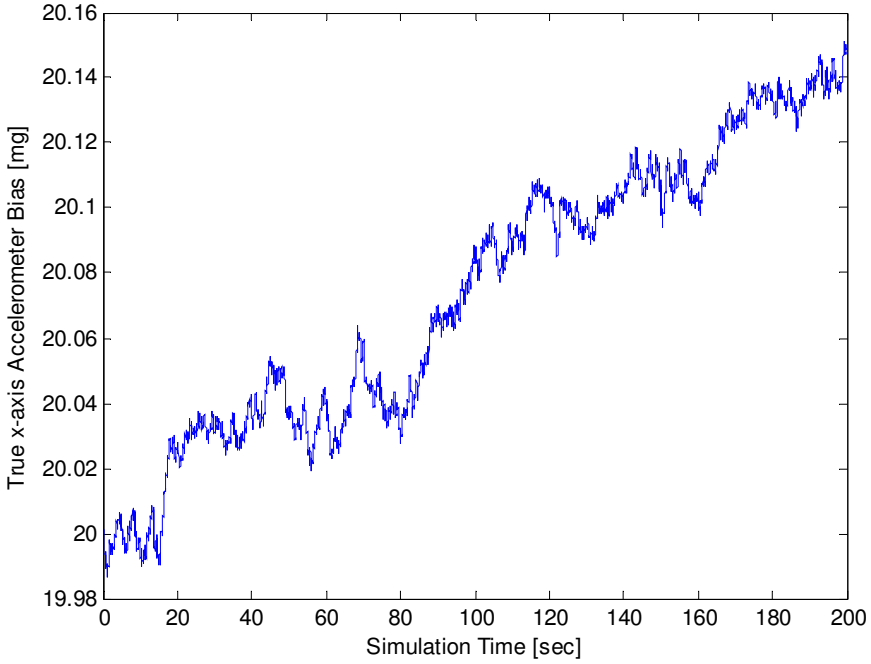

Figure 6. Random walk of x-axis accelerometer bias for 1-D case

The expected values (or standard deviation) for the estimate errors were taken after 100 seconds. The statistics for the 1-D model are given in table 3.

Table 3. Statistical data for truth and state estimations for 1-D case

Standard Deviation (SD) of GPS North position measurement error	$1m$
SD of North position estimation error	$0.15m$
SD of North velocity estimation error	$0.04m/s$
SD of x-axis accelerometer noise	$5mg$
SD of x-axis accelerometer bias-rate noise	$0.0001g/s$
True initial value of x-axis accelerometer bias	$20mg$

4.2 2-D Model

The 2-D motion utilizes two accelerometer measurements, two GPS position measurements and one angular rate measurement. The two accelerometer measurements available are along the body-fixed x and y axes. The two GPS position measurements are along the North and East axes. The angular rate measurement is about the body-fixed z axis. In order to use the body-fixed accelerometer and angular rate measurements, the Direction Cosine Matrix (DCM) is developed. This relates the orientation of the AGV in the ICS.

Before the DCM can be developed, the order of rotations must be defined. Even though this 2-D case has only one rotation, it will be necessary in the next section (3-D) to have a sound understanding of the remaining rotations. The first rotation is yaw. Yaw is the angle in the N-E plane from the North axis to the projection of the body-fixed x axis onto the N-E plane. The other rotations (pitch and roll) will be discussed in Section 4.3. Therefore, the DCM for the 2-D case is given by (4.10) (with yaw being ψ):

$$DCM_{ICS}^{BCS} = \begin{bmatrix} \cos(\psi) & \sin(\psi) & 0 \\ -\sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.10}$$

Note that the time notation has been removed but it is implied that each state fluctuates through time. A vector with components expressed in the ICS can be converted to components expressed in the BCS by multiplying it by DCM_{ICS}^{BCS}. For the reverse, the DCM must be transposed. In order to relate the body-fixed angular rates from the IMU to the Euler rates (yaw rate, pitch rate and roll rate) some additional work is required. For this 2-D case, it is trivial. The body-fixed angular rate about the z axis is the yaw rate, i.e.:

$$r = \dot{\psi} \tag{4.11}$$

For the 3-D case it will not be trivial. The step-by-step operation for the 3-D case will be presented in section 4.3. Once the Euler rates are known, they can be integrated to provide the Euler angles (yaw, pitch and roll).

For direction cosine matrices the inverse is the transpose because it is an orthogonal matrix. Therefore, the matrix DCM_{BCS}^{ICS} is the transpose of DCM_{ICS}^{BCS}, where the superscript

denotes the desired co-ordinate system, and the subscript the previous co-ordinate system. These matrices are used extensively throughout this thesis; therefore, it is essential to understand the notation. The characteristics and functions for the 2-D motion are listed in table 4 and equations (4.12) through (4.16).

Table 4. List of states/parameters for 2-D case

x_1	North position
x_2	East position
x_3	North velocity
x_4	East velocity
x_5	Yaw (ψ)
x_6	Bias in x-axis accelerometer
x_7	Bias in y-axis accelerometer
x_8	Bias in z-axis ARS
x_9	SFT for x-axis accelerometer
x_{10}	SFT for y-axis accelerometer
x_{11}	SFT for z-axis ARS

Model

$$\begin{gathered}\dot{x}_1 = x_3 \\ \dot{x}_2 = x_4 \\ \begin{bmatrix}\dot{x}_3 \\ \dot{x}_4\end{bmatrix} = DCM_{BCS}^{ICS}\begin{bmatrix} x_9\left(\tilde{\ddot{x}} - x_6\right) \\ x_{10}\left(\tilde{\ddot{y}} - x_7\right)\end{bmatrix} \\ \dot{x}_5 = x_{11}\left(\tilde{r} - x_8\right) \\ \dot{x}_{6-11} = 0\end{gathered} \tag{4.12}$$

Initial State Estimate Vector

$$\hat{\mathrm{x}}(t_0) = \begin{bmatrix}0 & 0 & 25 & 5 & 0 & 0 & 0 & 0 & 1 & 1 & 1\end{bmatrix}^T \tag{4.13}$$

Measurement and Process Noise Covariance Matrices

$$\begin{gathered}R_k = \mathrm{diag}\begin{bmatrix} v_{GPS,N}^2 & v_{GPS,E}^2\end{bmatrix} = \mathrm{diag}\begin{bmatrix} 1m^2 & 1m^2\end{bmatrix} \\ Q(t) = \mathrm{diag}\begin{bmatrix} w_{Acc,x}^2 & w_{Acc,y}^2 & w_r^2\end{bmatrix} \\ \therefore Q(t) = \mathrm{diag}\begin{bmatrix}(0.005g)^2 & (0.005g)^2 & (0.05\,{}^{\circ}\!/_{s})^2\end{bmatrix}\end{gathered} \tag{4.14}$$

Process Noise Mapping Matrix

$$G(t)=\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ x_9\cos(x_5) & -x_{10}\sin(x_5) & 0 \\ x_9\sin(x_5) & x_{10}\cos(x_5) & 0 \\ 0 & 0 & x_{11} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{4.15}$$

Initial Estimation Error Covariance Matrix

$$P_0=\begin{bmatrix} .5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & .5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & .5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & .5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & .005 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & .005 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & .005 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & .005 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .005 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .005 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .005 \end{bmatrix} \tag{4.16}$$

Or can be defined as:

$$P_0(\text{diag(1-4)})=0.5,\ P_0(\text{diag(5-11)})=0.005$$

To simulate the measurements, the truth was generated by maneuvering the vehicle in 2-D space. There were three maneuvers that were tested. The first used random numbers for the two accelerations. The second modeled the AGV driving in a circle. For the final maneuver the vehicle path was a sinusoid with the function shown in equation (4.17).

$$N(t) = 200\sin\left(2\pi\frac{E}{250}\right)$$
$$E(t) = 5t \tag{4.17}$$

There are 15 figures for the 2-D case because of the addition of an accelerometer measurement, an ARS measurement and a GPS position measurement. Each of these increases the complexity of the system, which is one reason for discussing each of the cases separately, as opposed to displaying the 3-D case only. Figure 7 shows the true path of the AGV in the N-E plane. Figure 8 displays the North position estimation error. Figure 9 displays the East position estimation error. Figure 10 displays the error of the North velocity estimate. Figure 11 displays the error of the East velocity estimate. Figures 12 and 13 display the errors of the x and y-axis accelerometer bias estimates, respectively. Figure 14 shows the error of the z-axis ARS bias estimate. Figure 15 shows how well the yaw angle was estimated. Figures 16, 17 and 18 show the percentage error of the SFT estimates for the three IMU measurements. Finally, Figures 19, 20 and 21 demonstrate the random walk of the three IMU measurements that were compensated for by the KF. Also, table 5 includes the statistics of the simulation (both truth and estimates).

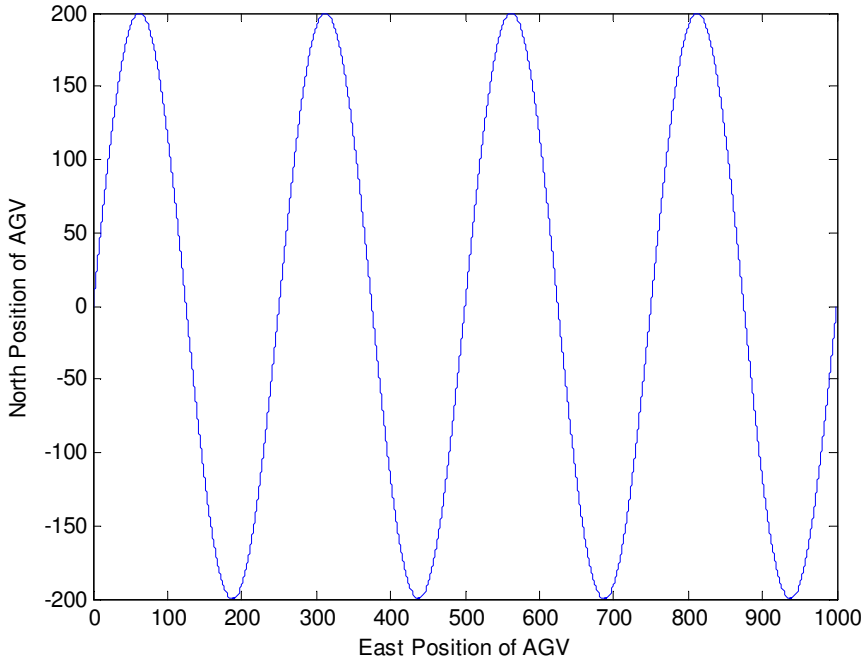

Figure 7. True motion of AGV in N-E plane for 2-D and 3-D cases

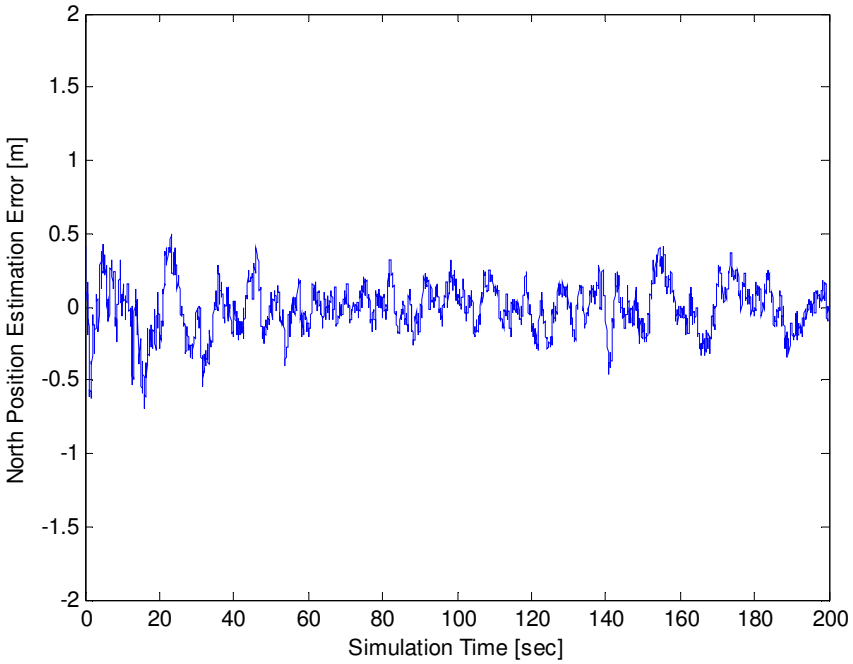

Figure 8. North position estimation error for 2-D case

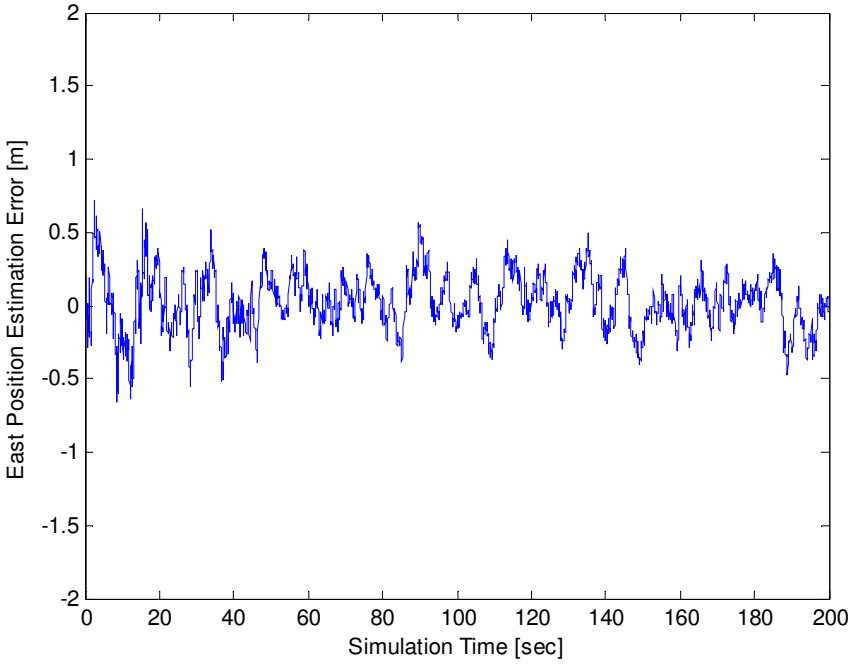

Figure 9. East position estimation error for 2-D case

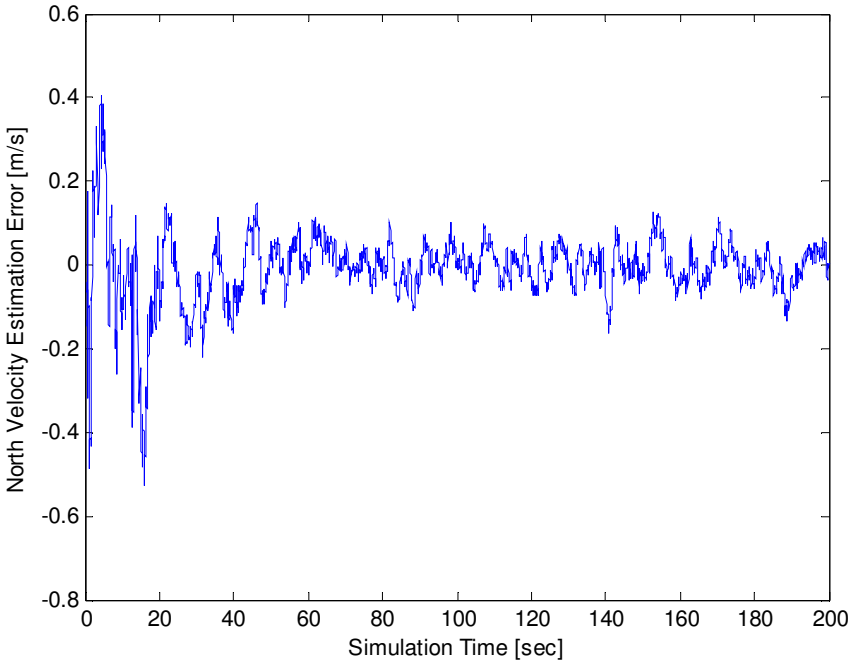

Figure 10. North velocity estimation error for 2-D case

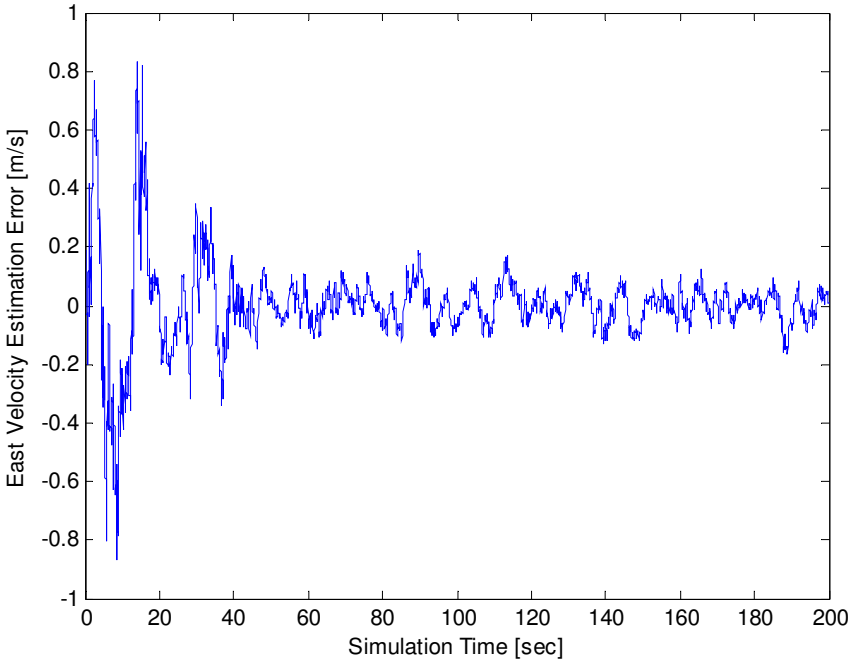

Figure 11. East velocity estimation error for 2-D case

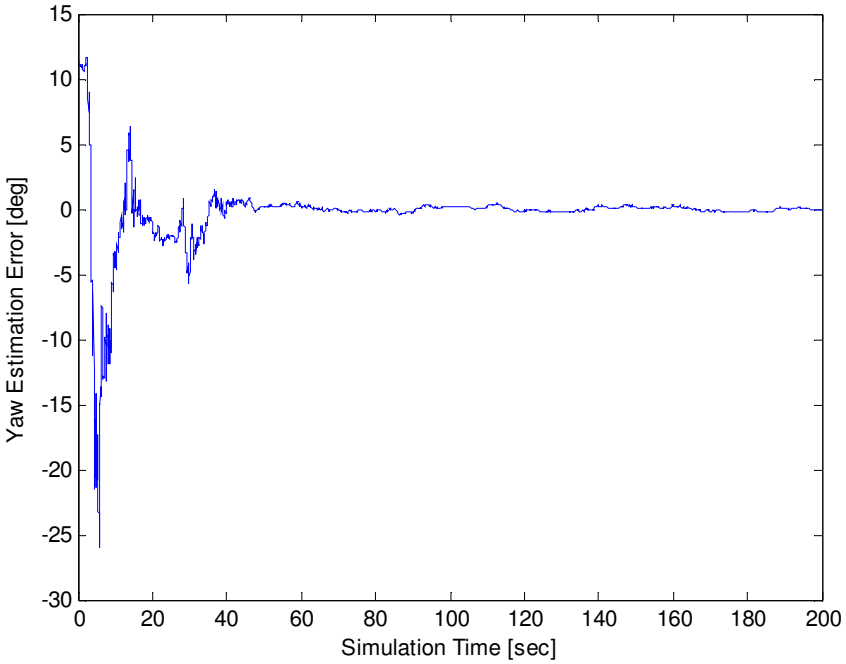

Figure 12. Yaw estimation error for 2-D case

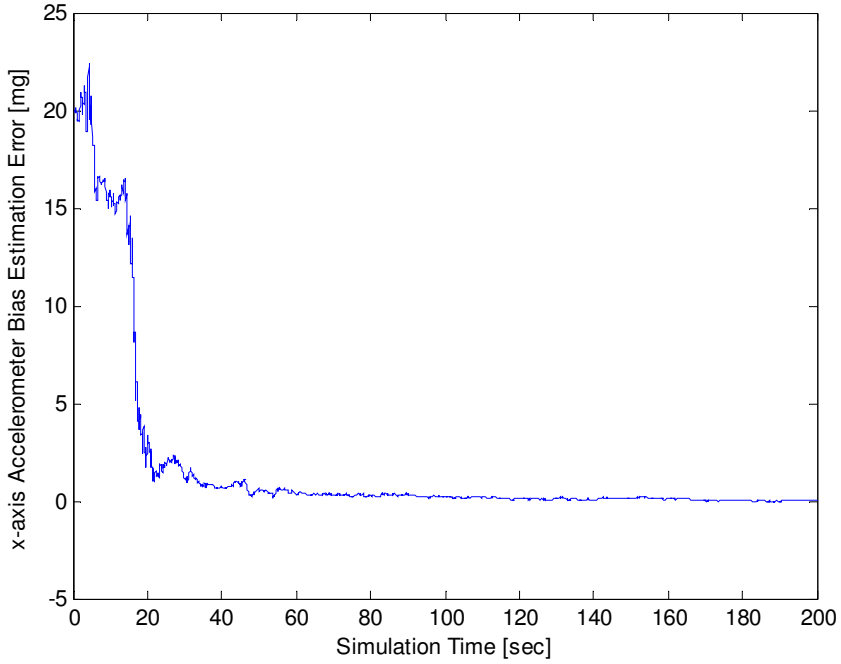

Figure 13. x-axis accelerometer bias estimation error for 2-D case

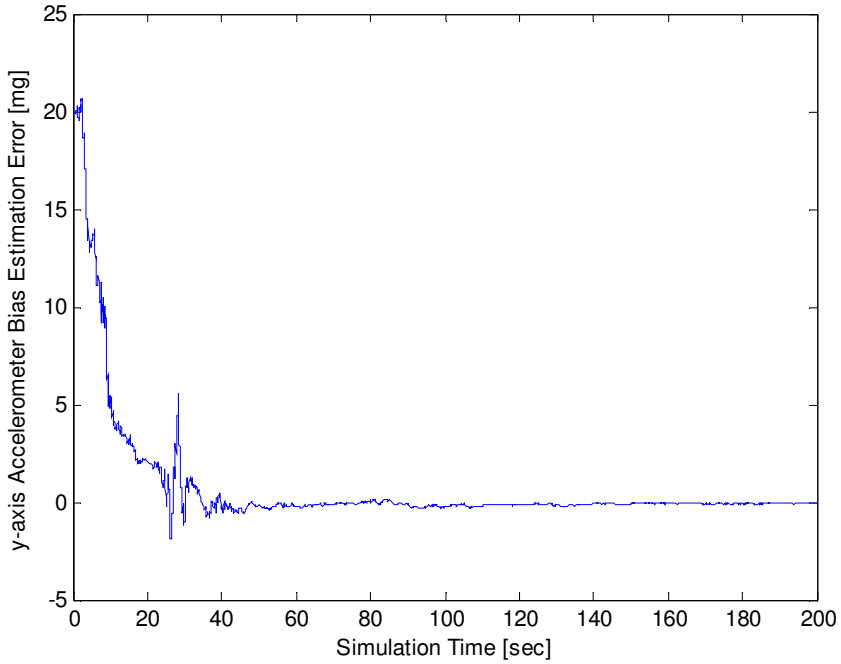

Figure 14. y-axis accelerometer bias estimation error for 2-D case\

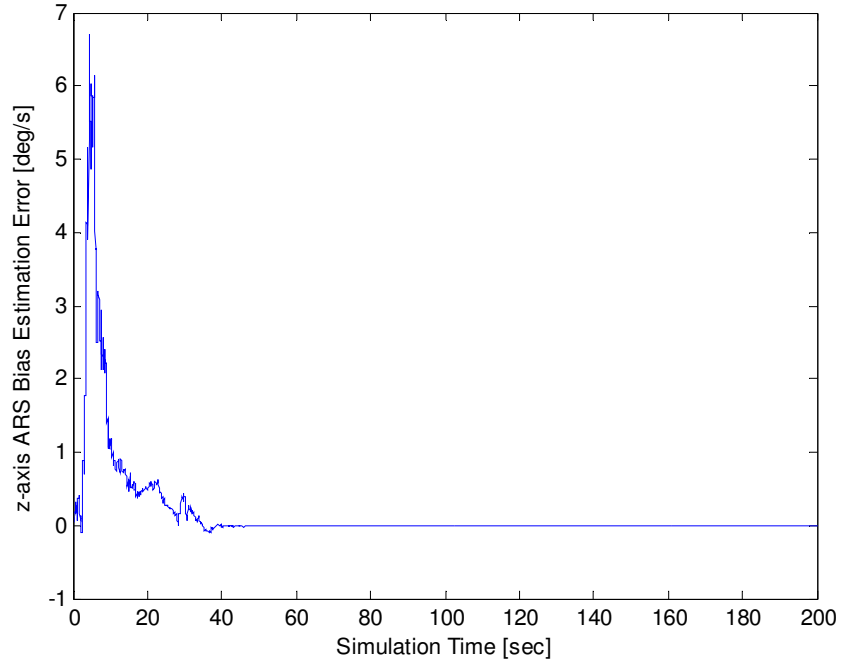

Figure 15. z-axis ARS bias estimation error for 2-D case

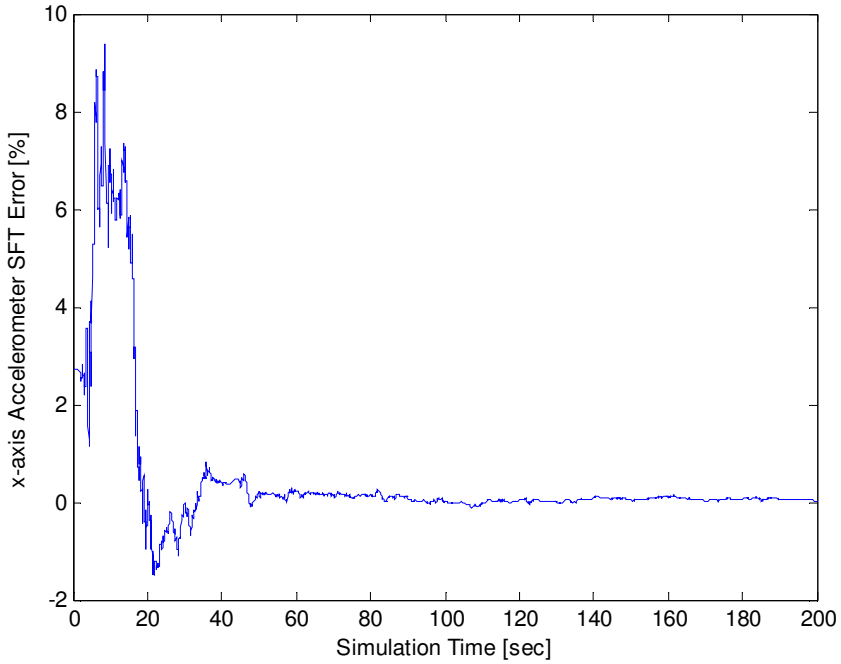

Figure 16. x-axis accelerometer SFT percentage error for 2-D case

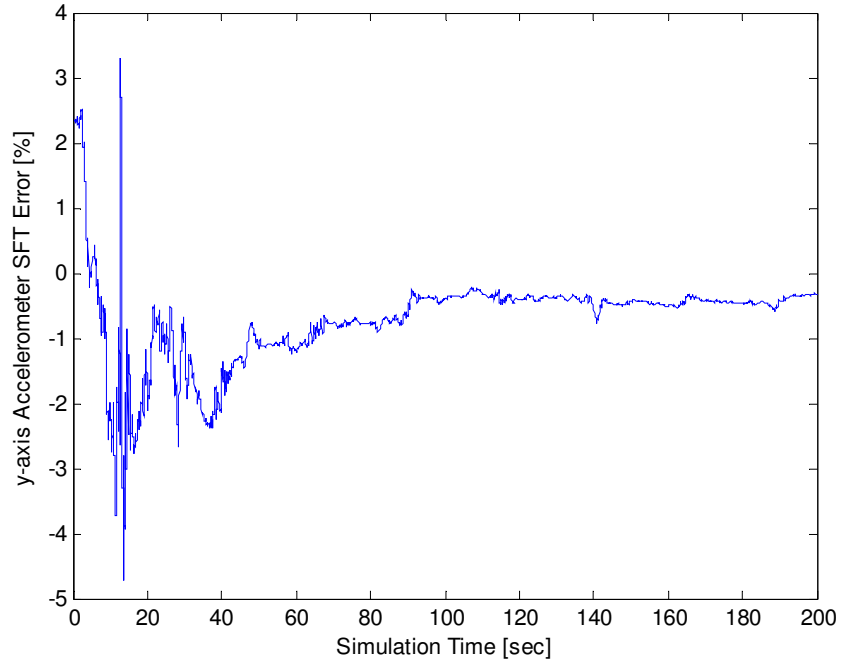

Figure 17. y-axis accelerometer SFT percentage error for 2-D case

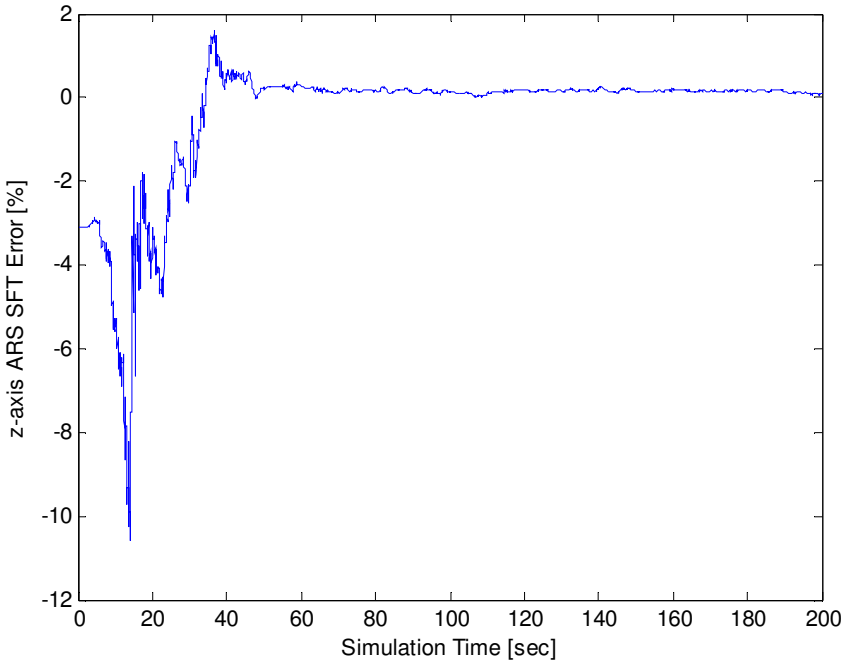

Figure 18. z-axis ARS SFT percentage error for 2-D case

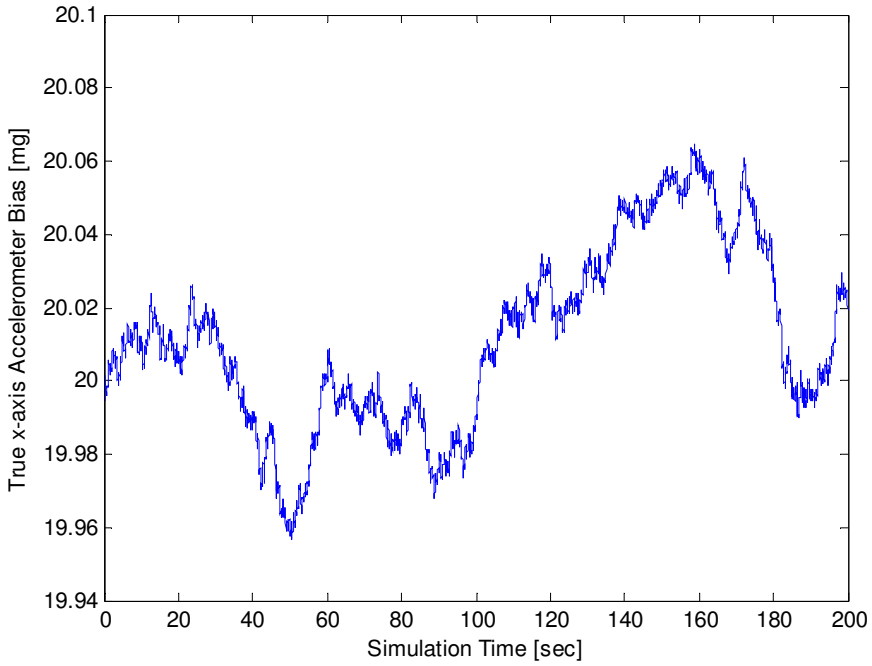

Figure 19. Random walk of x-axis accelerometer bias for 2-D case

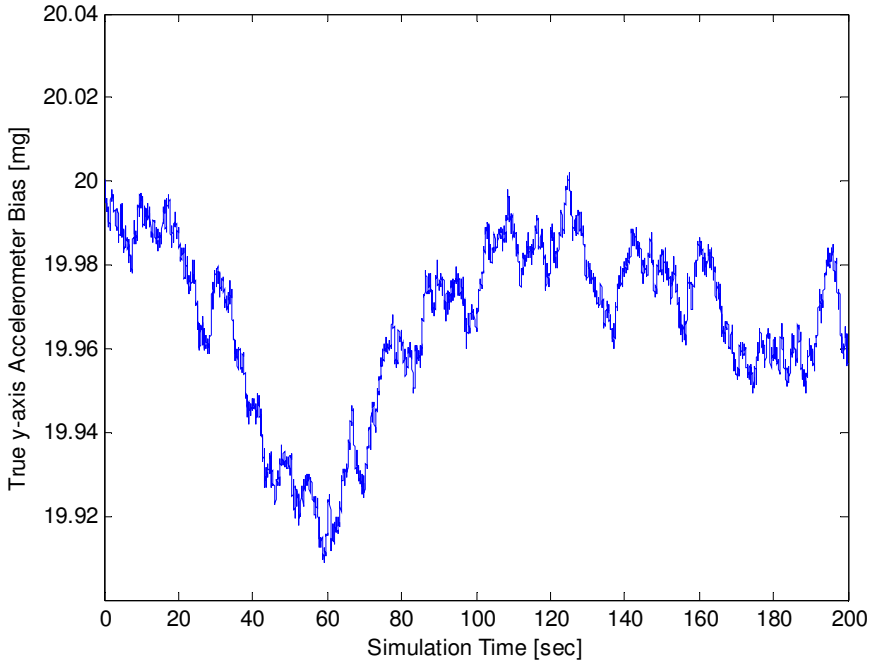

Figure 20. Random walk of y-axis accelerometer bias for 2-D case

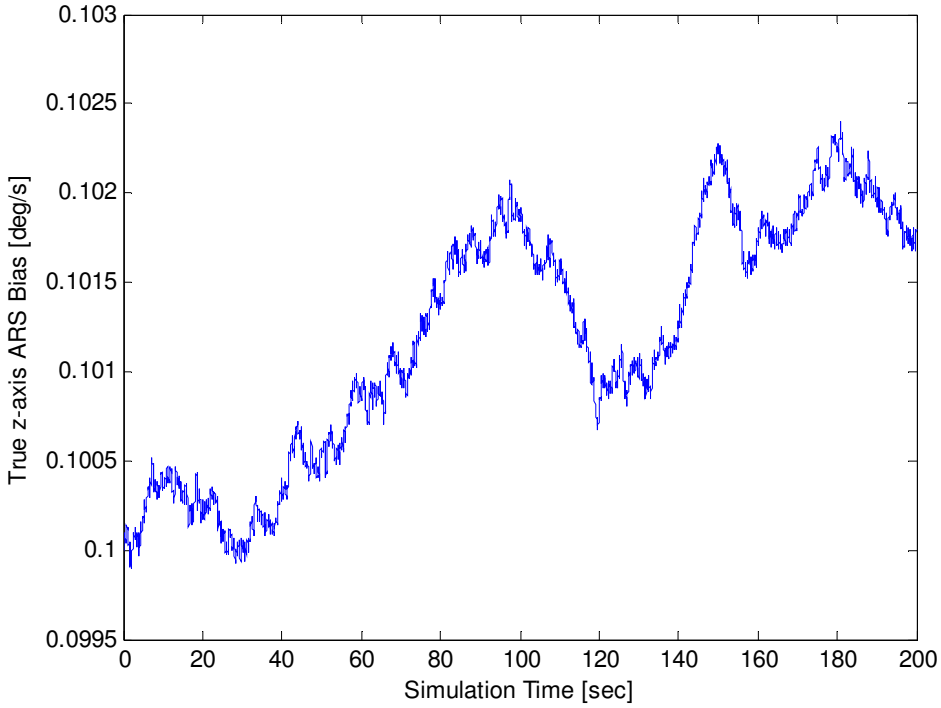

Figure 21. Random walk of z-axis ARS bias for 2-D case

Table 5. Statistical data for truth and state estimations for 2-D case

SD of North and East position measurement errors	$1m$
SD of North and East position estimation errors	$0.15m$
SD of North and East velocity estimation errors	$0.05m/s$
SD of Yaw estimation error	$0.15°$
SD of x,y-axes accelerometers noises	$5mg$
SD of x,y-axes accelerometers bias-rate noises	$0.0001g/s$
True initial value of x,y-axes accelerometers biases	$2mg$
SD of z-axis ARS noise	$0.05°/s$
SD of z-axis ARS bias-rate noise	$0.002°/s^2$
True initial value z-axis ARS bias	$0.1°/s$

4.3 3-D Model

The general model has six degrees of freedom. The AGV can translate in the North, East and Down directions, as well as rotate about the x, y and z axes. The measurements available are the three GPS position measurements (N, E and D), the three accelerometer (x, y and z) measurements and three ARS measurements (p, q and r). The DCM for the general case is much more complex than the two-dimensional case because the AGV can pitch and roll for the 3-D case. The order of rotation is yaw (ψ), pitch (θ) and then roll (ϕ). Thus, the DCM for general motion, using the order specified is the following:

$$DCM_{ICS}^{BCS} = A_{roll} A_{pitch} A_{yaw} \tag{4.18}$$

Where:

$$\begin{aligned} A_{roll} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & \sin(\phi) \\ 0 & -\sin(\phi) & \cos(\phi) \end{bmatrix} \\ A_{pitch} &= \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \\ A_{yaw} &= \begin{bmatrix} \cos(\psi) & \sin(\psi) & 0 \\ -\sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{4.19}$$

Therefore,

$$DCM_{ICS}^{BCS} = \begin{bmatrix} c\psi c\theta & s\psi c\theta & -s\theta \\ c\psi s\theta s\phi - s\psi c\phi & s\psi s\theta s\phi + c\psi c\phi & c\theta s\phi \\ \mathrm{c}\psi\,\mathrm{s}\,\theta\,\mathrm{c}\,\phi + s\psi s\phi & s\psi s\theta c\phi - c\psi s\phi & c\theta c\phi \end{bmatrix} \tag{4.20}$$

Where: $c\psi = \cos(\psi),\ s\psi = \sin(\psi)$, etc…

In order to acquire the Euler rates from the body-fixed angular rates (p, q and r), a step-by-step analysis using the order of the rotations must be performed. The following demonstrates this process:

Step 1: Determine the order of rotations and the axes about which they rotate

$$\begin{aligned} &\psi \text{ about D axis} \\ &\theta \text{ about E' axis} \\ &\phi \text{ about N" axis} \end{aligned} \tag{4.21}$$

Step 2: Determine the body-fixed angular velocities for each of the rotations

$$\begin{aligned} \omega' &= \{0\}N' + \{0\}E' + \{\dot{\psi}\}D' \\ \omega'' &= \{-\dot{\psi}\sin(\theta)\}N'' + \{\dot{\theta}\}E'' + \{\dot{\psi}\cos(\theta)\}D'' \\ \omega''' &= \omega_{BCS} = \{p\}x + \{q\}y + \{r\}z \\ p &= \dot{\phi} - \dot{\psi}\sin(\theta) \\ q &= \dot{\theta}\cos(\phi) + \dot{\psi}\cos(\theta)\sin(\phi) \\ r &= -\dot{\theta}\sin(\phi) + \dot{\psi}\cos(\theta)\cos(\phi) \end{aligned} \tag{4.22}$$

Where: $\omega' = \{a\}N' + ...$ denotes that "a" is the scalar value of the N' component of the vector ω'

Step 3: Put solution into matrix format

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} -s\theta & 0 & 1 \\ c\theta s\phi & c\phi & 0 \\ c\theta c\phi & -s\phi & 0 \end{bmatrix} \begin{bmatrix} \dot{\psi} \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix} \tag{4.23}$$

Step 4: Invert the 3x3 matrix to obtain the Euler Rates

$$\begin{bmatrix} \dot{\psi} \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} 0 & \frac{s\phi}{c\theta} & \frac{c\phi}{c\theta} \\ 0 & c\phi & -s\phi \\ 1 & t\theta s\phi & t\theta c\phi \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{4.24}$$

Where: $t\theta = \tan(\theta)$

Once the Euler rates are available, they can be integrated (using initial conditions) to obtain the Euler angles. Several elements of the transformation matrix of equation (4.24) are undefined for $\theta = \pm 90°$ which corresponds to the AGV pitching $\pm 90°$ which will not occur for normal operations. The AGV may roll or yaw to any position and not cause any mathematical singularities.

The general motion consists of 21 states/parameters. The first nine components are the system states which include North, East and Down positions/velocities, as well as the yaw, pitch and roll angles. The next 12 components are the system parameters which include the biases and SFT's for the accelerometers and ARS's along and about the x, y and z axes, respectively. The definitions are in table 6.

Table 6. List of states/parameters for 3-D case

x_1	North position
x_2	East position
x_3	Down position
x_4	North velocity
x_5	East velocity
x_6	Down velocity
x_7	Yaw (ψ)
x_8	Pitch (θ)
x_9	Roll (ϕ)
x_{10}	Bias in x-axis accelerometer
x_{11}	Bias in y-axis accelerometer
x_{12}	Bias in z-axis accelerometer
x_{13}	Bias in x-axis ARS
x_{14}	Bias in y-axis ARS
x_{15}	Bias in x-axis ARS
x_{16}	SFT for x-axis accelerometer
x_{17}	SFT for y-axis accelerometer
x_{18}	SFT for z-axis accelerometer
x_{19}	SFT for x-axis ARS
x_{20}	SFT for y-axis ARS
x_{21}	SFT for z-axis ARS

With the general motion, the gravity model must now be included. The accelerometers will sense and output the gravitational component which must be compensated for by the KF. Therefore, the general motion accelerometer models (and the ARS models) are the following:

$$\begin{aligned}
\ddot{x} &= x_{16}\left(\tilde{\ddot{x}} - x_{10}\right) - g\sin(x_8) \\
\ddot{y} &= x_{17}\left(\tilde{\ddot{y}} - x_{11}\right) + g\cos(x_8)\sin(x_9) \\
\ddot{z} &= x_{18}\left(\tilde{\ddot{z}} - x_{12}\right) + g\cos(x_8)\cos(x_9) \\
p &= x_{19}\left(\tilde{p} - x_{13}\right) \\
q &= x_{20}\left(\tilde{q} - x_{14}\right) \\
r &= x_{21}\left(\tilde{r} - x_{15}\right)
\end{aligned} \tag{4.25}$$

The KF has to compensate for six biases, six SFT's and three gravitational components while the system is dynamic. This is why it is an extraordinary filter. As the system has become more complex, it becomes more sensitive to noise and inaccuracies. The functions used by the KF are shown in equations (4.24) through (4.28).

Model

$$\begin{aligned}
\dot{x}_1 &= x_4 \\
\dot{x}_2 &= x_5 \\
\dot{x}_3 &= x_6 \\
\begin{bmatrix} \dot{x}_4 \\ \dot{x}_5 \\ \dot{x}_6 \end{bmatrix} &= DCM_{BCS}^{ICS} \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} \\
\dot{x}_7 &= \frac{1}{\cos(x_8)}(q\sin(x_9) + r\cos(x_9)) \\
\dot{x}_8 &= q\cos(x_9) - r\sin(x_9) \\
\dot{x}_9 &= p + \tan(x_8)(q\sin(x_9) + r\cos(x_9)) \\
\dot{x}_{10-21} &= 0
\end{aligned} \tag{4.26}$$

Initial State Estimate Vector

$$\begin{aligned}
\hat{\mathrm{x}}(t_0, 1-9) &= \begin{bmatrix} 0 & 0 & 0 & 25 & 5 & 0 & 0 & 0 & 0 \end{bmatrix}^T \\
\hat{\mathrm{x}}(t_0, 10-21) &= \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}^T
\end{aligned} \tag{4.27}$$

Measurement and Process Noise Covariance Matrices

$$
\begin{gathered}
R_k = \operatorname{diag}\begin{bmatrix} v_{GPS,N}^2 & v_{GPS,E}^2 & v_{GPS,D}^2 \end{bmatrix} \\
v_{GPS,N}^2 = v_{GPS,E}^2 = v_{GPS,D}^2 = 1m^2 \\
Q(t) = \operatorname{diag}\begin{bmatrix} w_{Acc,x}^2 & w_{Acc,y}^2 & w_{Acc,z}^2 & w_p^2 & w_q^2 & w_r^2 \end{bmatrix} \\
w_{Acc,x}^2 = w_{Acc,y}^2 = w_{Acc,z}^2 = (0.005g)^2,\ w_p^2 = w_q^2 = w_r^2 = (0.05\,{}^{\circ}\!/_{s})^2
\end{gathered}
\tag{4.28}
$$

Process Noise Mapping Matrix

$$
G(t) = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
G_{41} & G_{42} & G_{43} & 0 & 0 & 0 \\
G_{51} & G_{52} & G_{53} & 0 & 0 & 0 \\
G_{61} & G_{62} & G_{63} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \dfrac{sx_9}{cx_8}x_{20} & \dfrac{sx_9}{cx_8}x_{21} \\
0 & 0 & 0 & 0 & cx_9x_{20} & -sx_9x_{21} \\
0 & 0 & 0 & x_{19} & tx_8sx_9x_{20} & tx_8cx_9x_{21} \\
 & & & 0(12\text{x}6) & &
\end{bmatrix}
\tag{4.29}
$$

Where:

$$
\begin{bmatrix} G_{41} & G_{42} & G_{43} \\ G_{51} & G_{52} & G_{53} \\ G_{61} & G_{62} & G_{63} \end{bmatrix} = DCM_{BCS}^{ICS} \begin{bmatrix} x_{16} & 0 & 0 \\ 0 & x_{17} & 0 \\ 0 & 0 & x_{18} \end{bmatrix}
$$

Initial Estimation Error Covariance Matrix

$$
\begin{gathered}
P_0(\operatorname{diag}(1-6)) = 0.5,\ P_0(\operatorname{diag}(7-12,16-18)) = 0.05 \\
P_0(\operatorname{diag}(13-15,19-21)) = 0.005
\end{gathered}
\tag{4.30}
$$

Simulating the GPS North and East positions was based on the same maneuver as in the 2-D case. The GPS Down position as well as the pitch and roll angles were based on sinusoidal motions shown in equation (4.31). Refer to figure 22 for the true Down position as a function of time. Figures 23 and 24 are the true pitch and roll angles of the AGV. Several different combinations of frequencies and amplitudes were used to reasonably simulate the pitching and rolling of the AGV.

$$\begin{aligned} D(t) &= -4.5\cos\left(\frac{t}{3}\right) + 5\cos\left(\frac{t}{5}\right) \\ \theta(t) &= -1.2\cos\left(\frac{t}{2}\right) - 3.6\sin\left(\frac{t}{3}\right) \\ \phi(t) &= -1.2\cos(t) - 3.6\sin\left(\frac{t}{2}\right) \end{aligned} \tag{4.31}$$

The solution for the general motion is presented in 30 figures. Figures 25 through 27 demonstrate both the stability and the accuracy of the North, East and Down position estimates from the KF. Figures 28 through 30 show the respective North, East and Down velocity estimation errors. Figures 31 through 33 represent the errors in the Euler angle estimates. Notice that yaw is the least accurate of the three. This is because the yaw is the least observable of the three angles. If a magnetometer were used to observe the yaw directly, the yaw estimate would be as good as the pitch and roll estimates, or better. Figures 34 through 39 show how the KF is estimating the acceleration and ARS biases, which are randomly walking. Figures 40 through 45 show the SFT estimate percentage errors for all six IMU measurements. Figures 46 through 51 represent the true accelerometer and ARS biases to give an idea of how much they are walking. Also, table 7 includes the statistical information for the results.

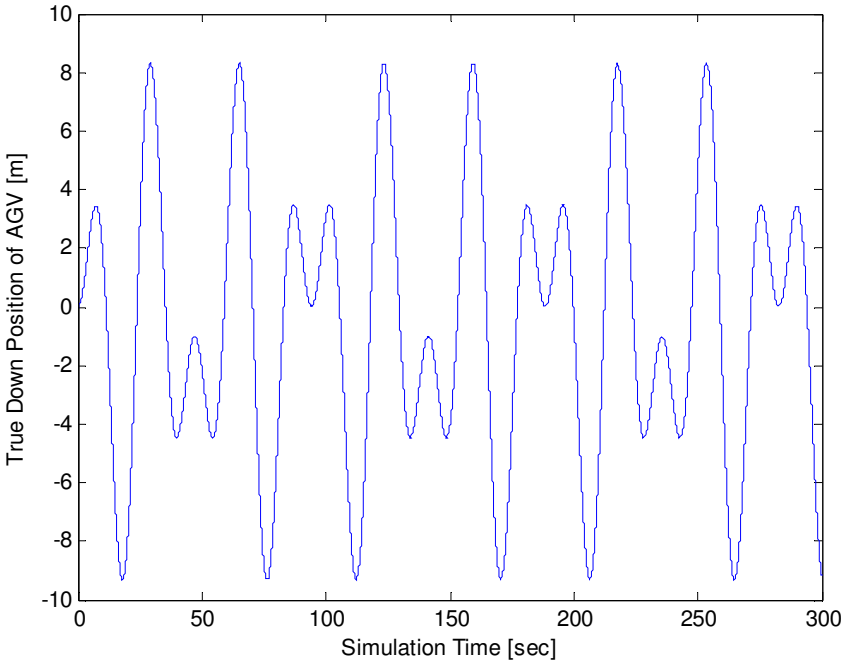

Figure 22. True Down position of AGV

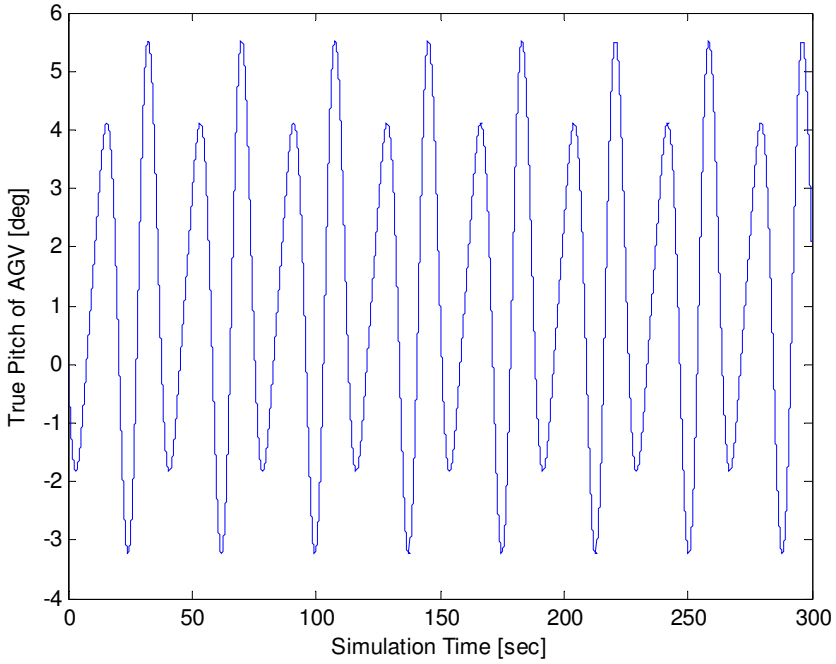

Figure 23. True pitch of AGV

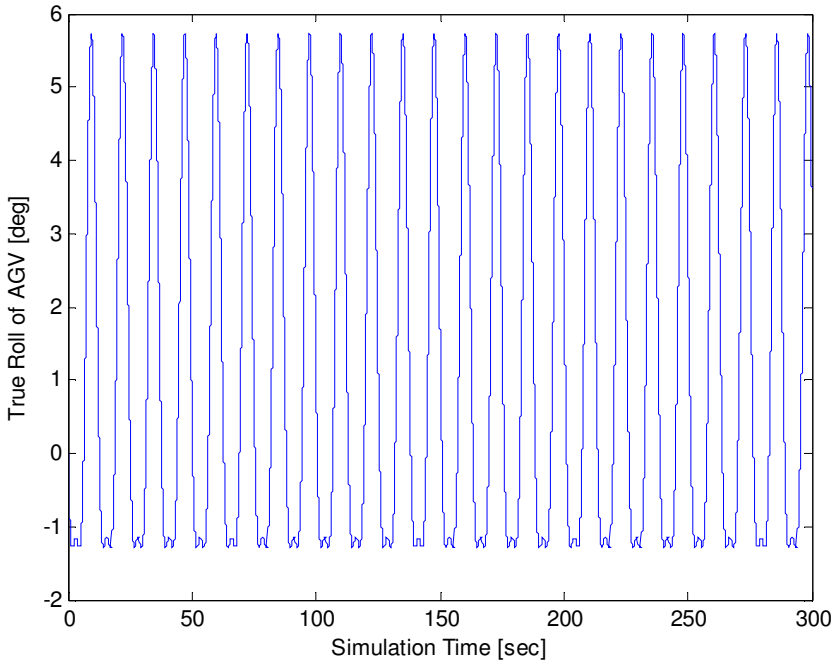

Figure 24. True roll of AGV

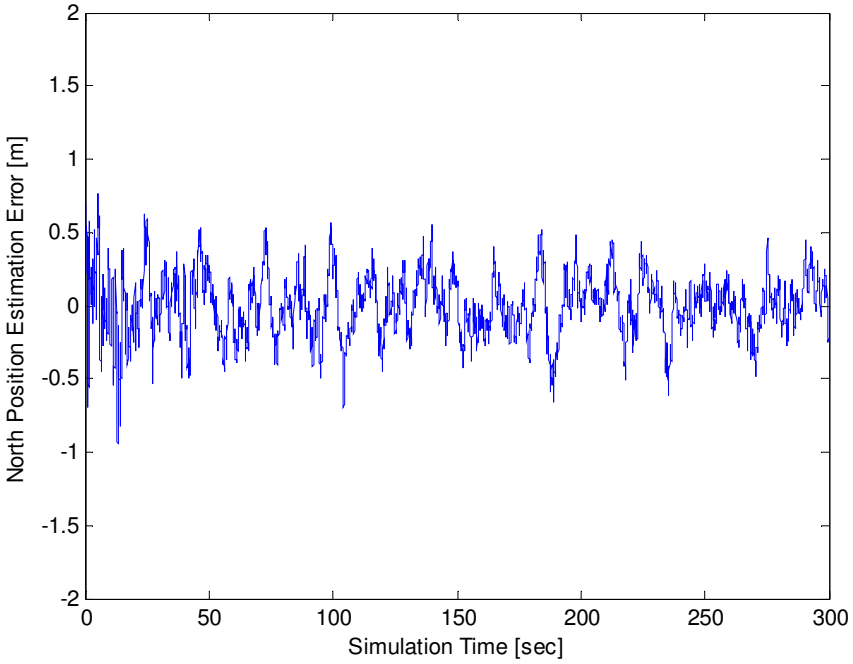

Figure 25. North position estimation error for 3-D case

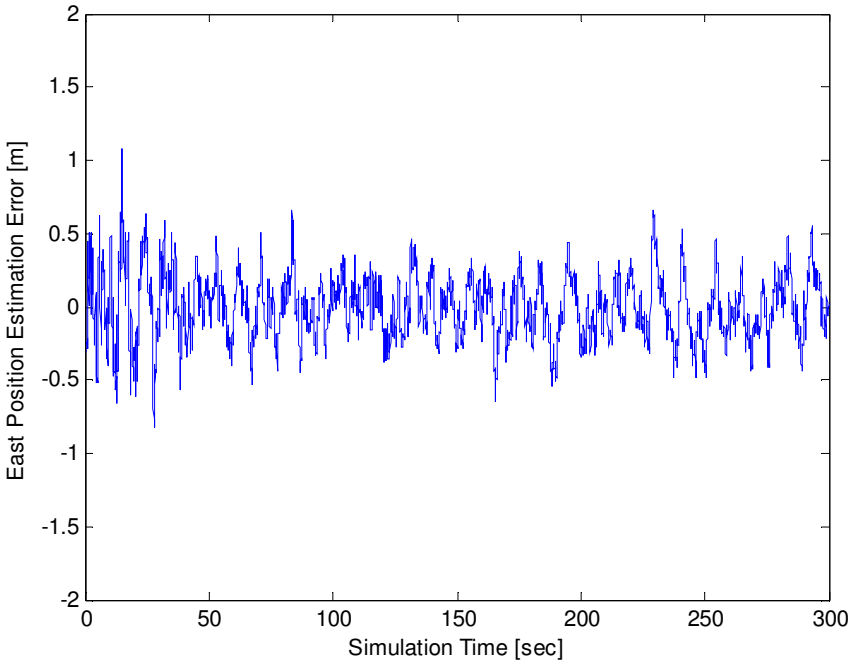

Figure 26. East position estimation error for 3-D case

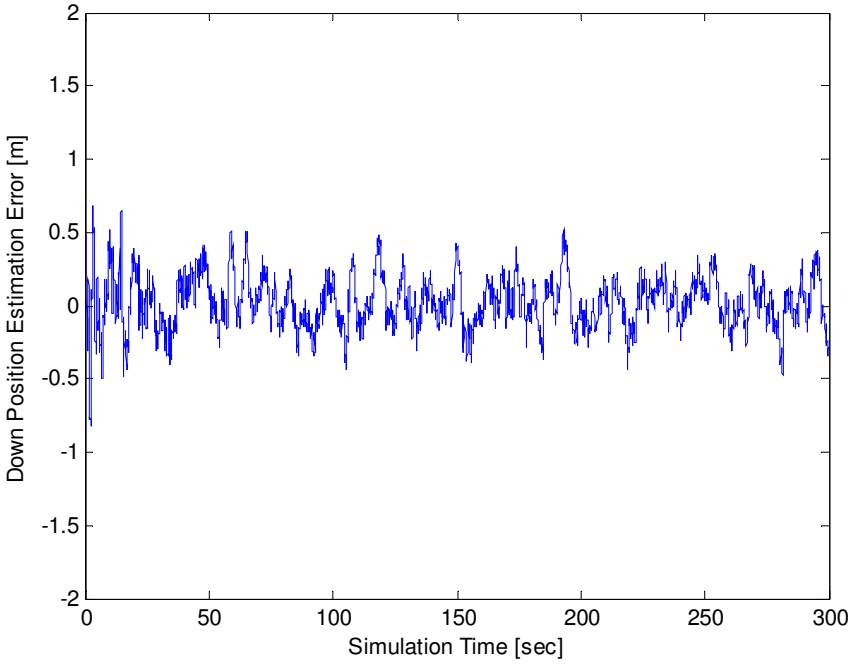

Figure 27. Down position estimation error

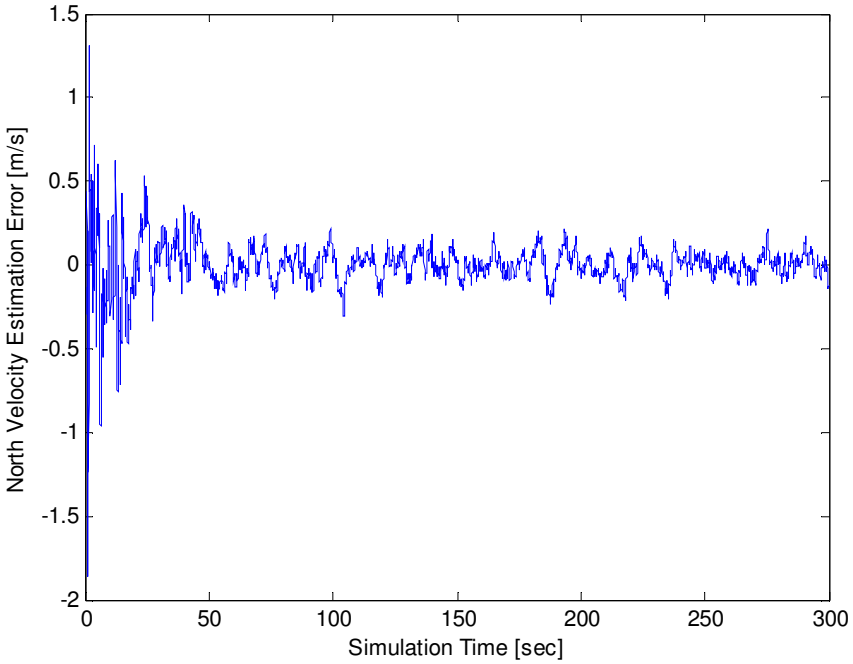

Figure 28. North velocity estimation error for 3-D case

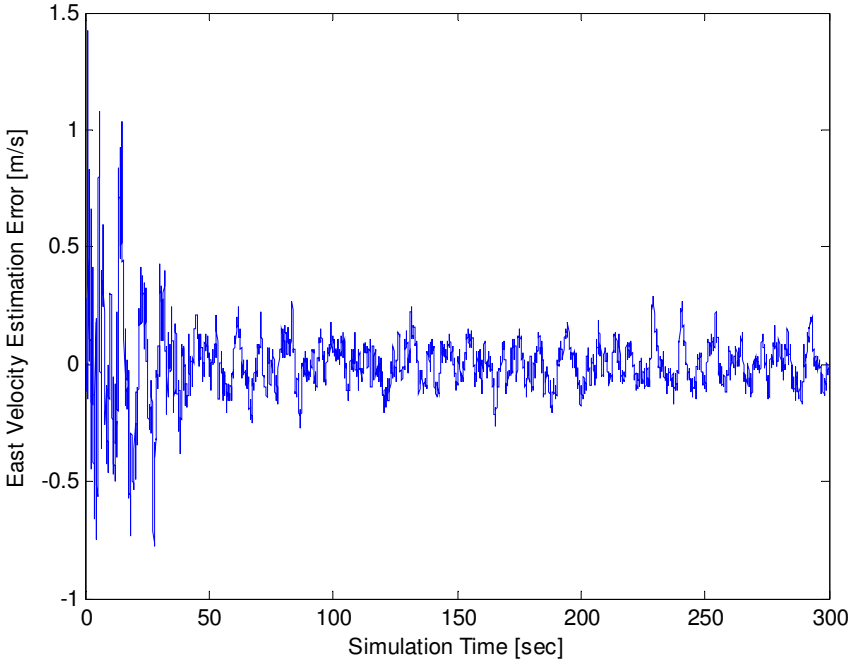

Figure 29. East velocity estimation error for 3-D case

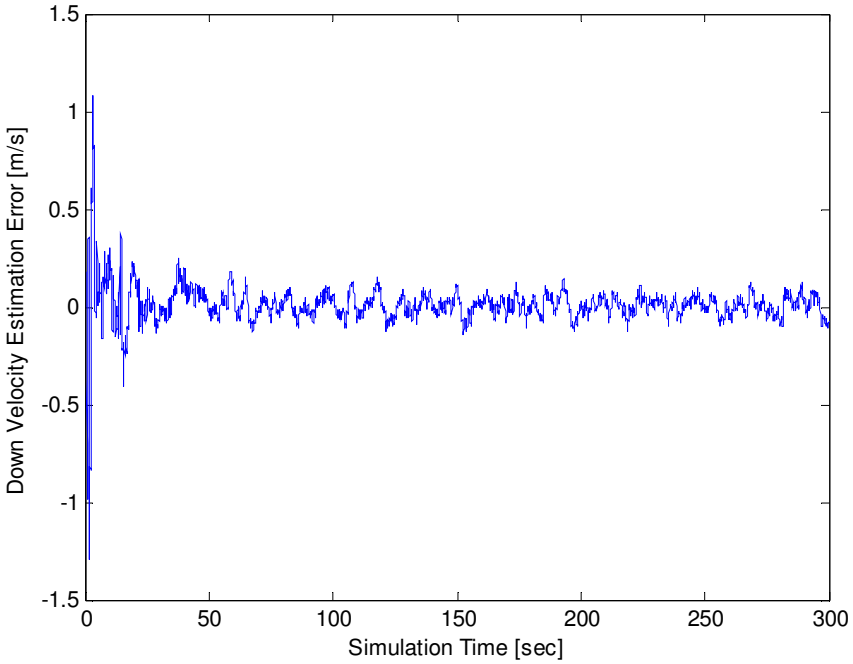

Figure 30. Down velocity estimation error

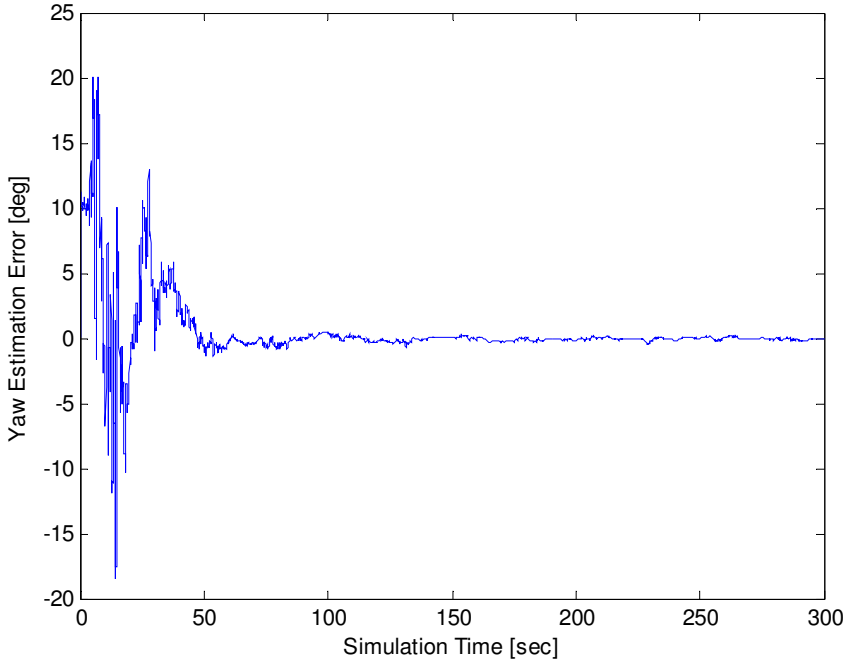

Figure 31. Yaw estimation error for 3-D case

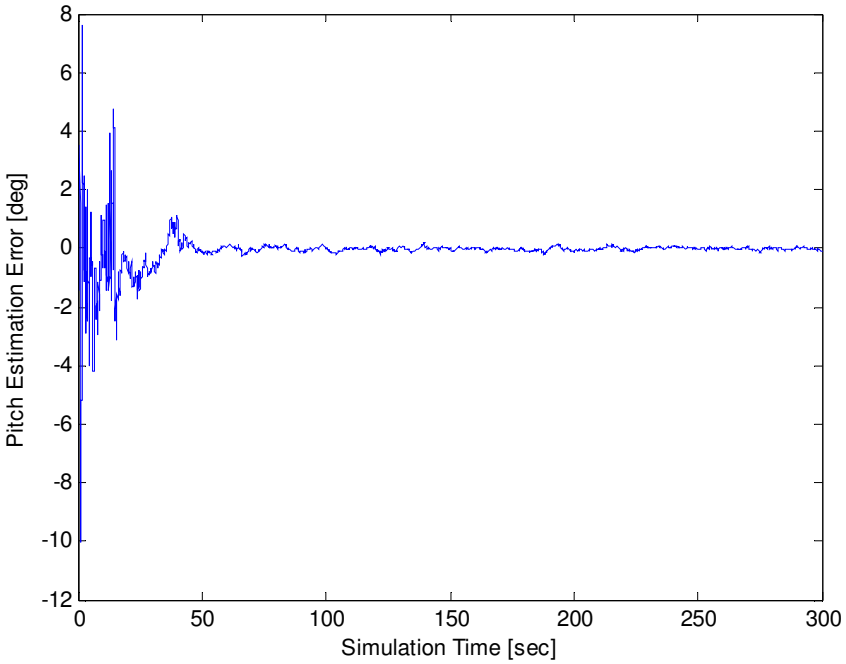

Figure 32. Pitch estimation error

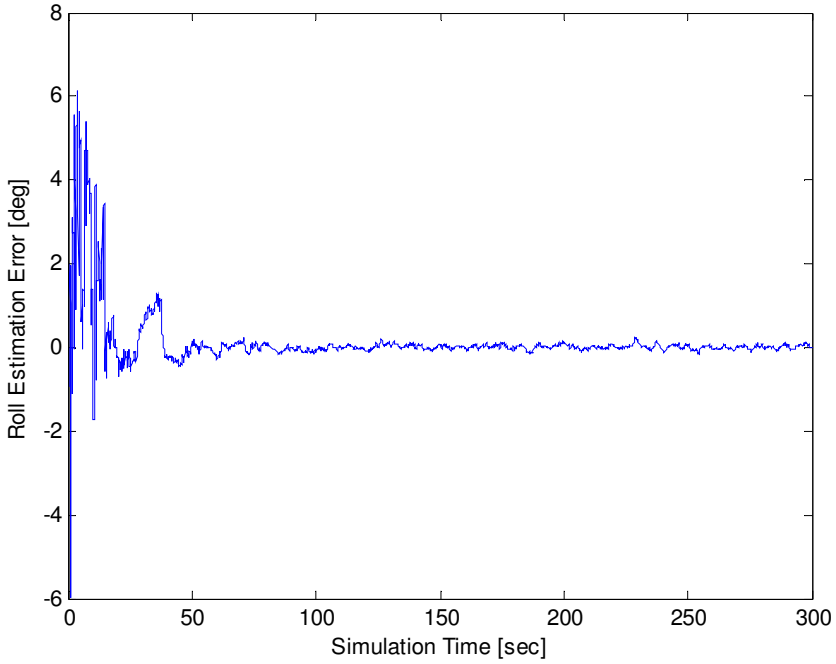

Figure 33. Roll estimation error

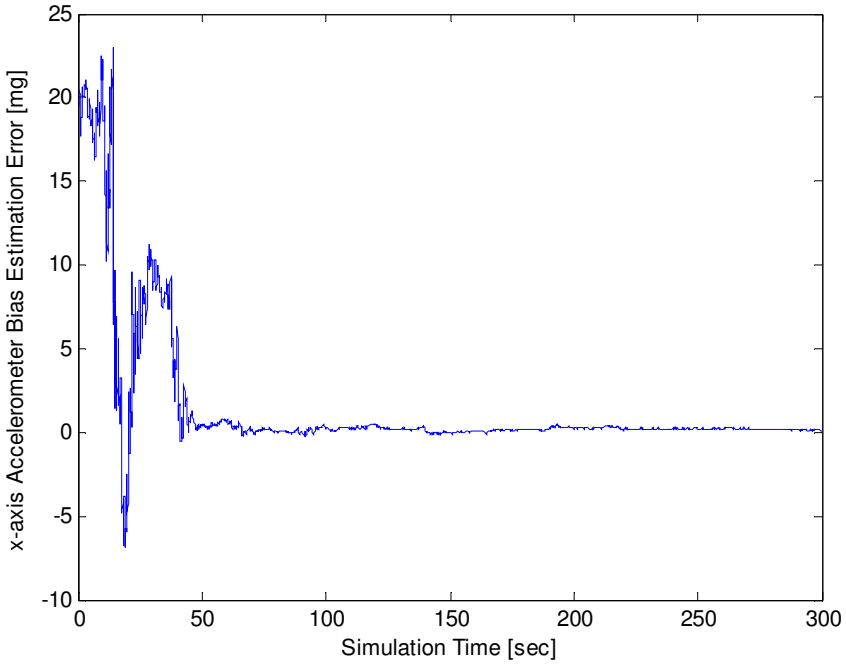

Figure 34. x-axis accelerometer bias estimation error for 3-D case

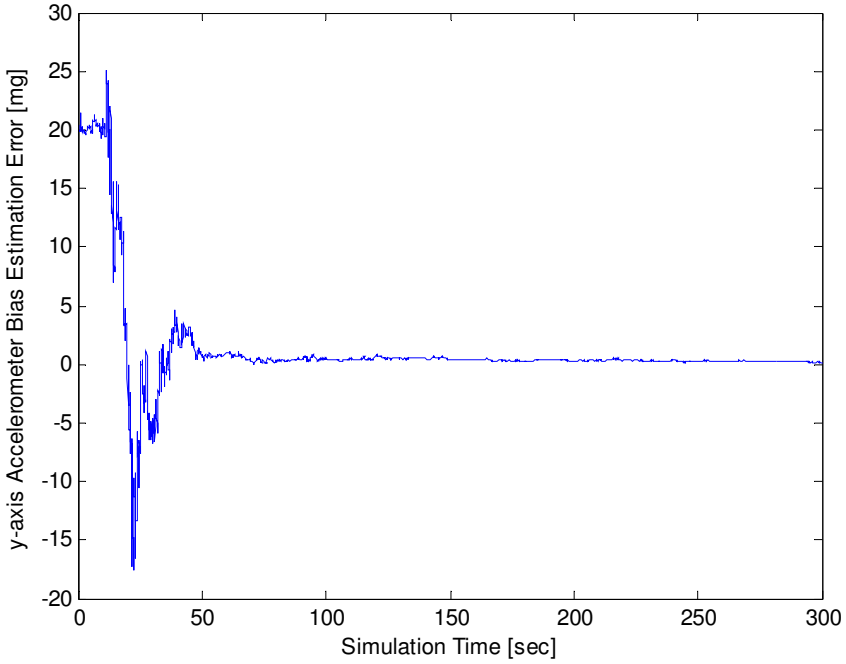

Figure 35. y-axis accelerometer bias estimation error for 3-D case

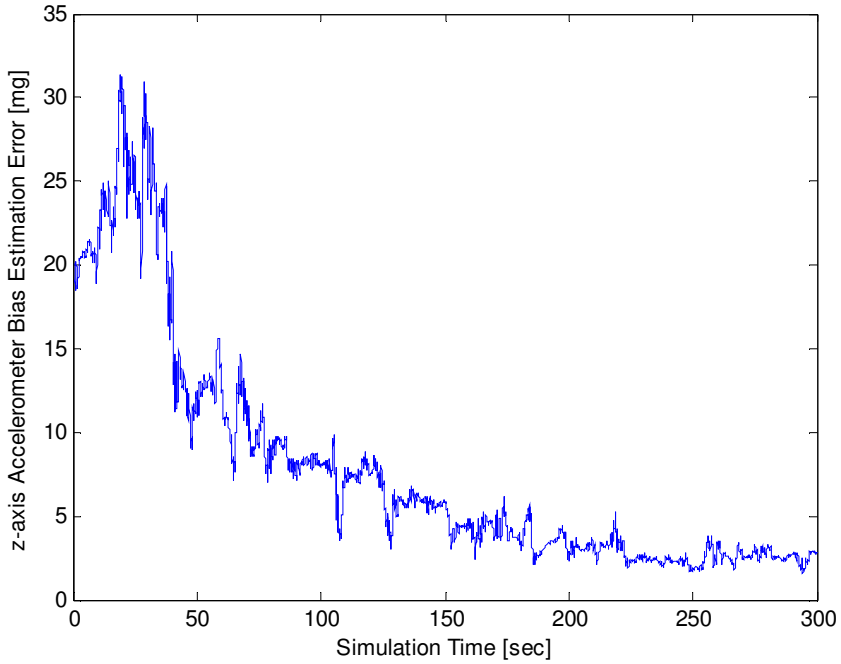

Figure 36. z-axis accelerometer bias estimation error

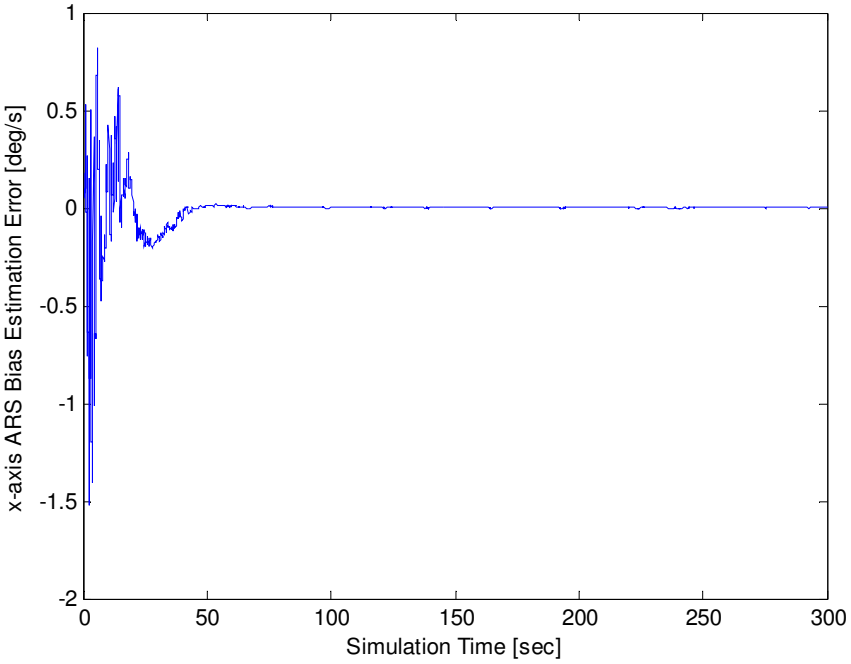

Figure 37. x-axis ARS bias estimation error

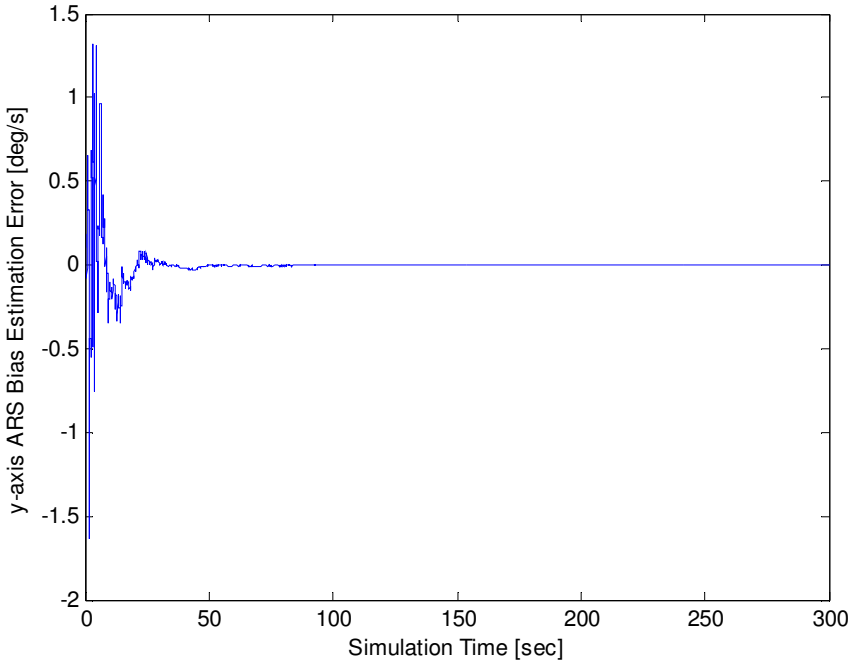

Figure 38. y-axis ARS bias estimation error

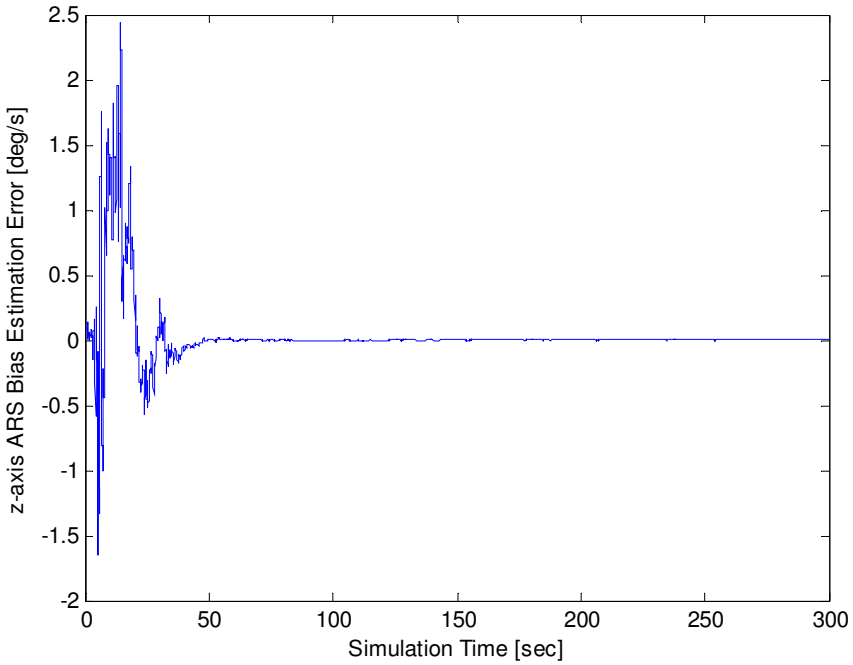

Figure 39. z-axis ARS bias estimation error for 3-D case

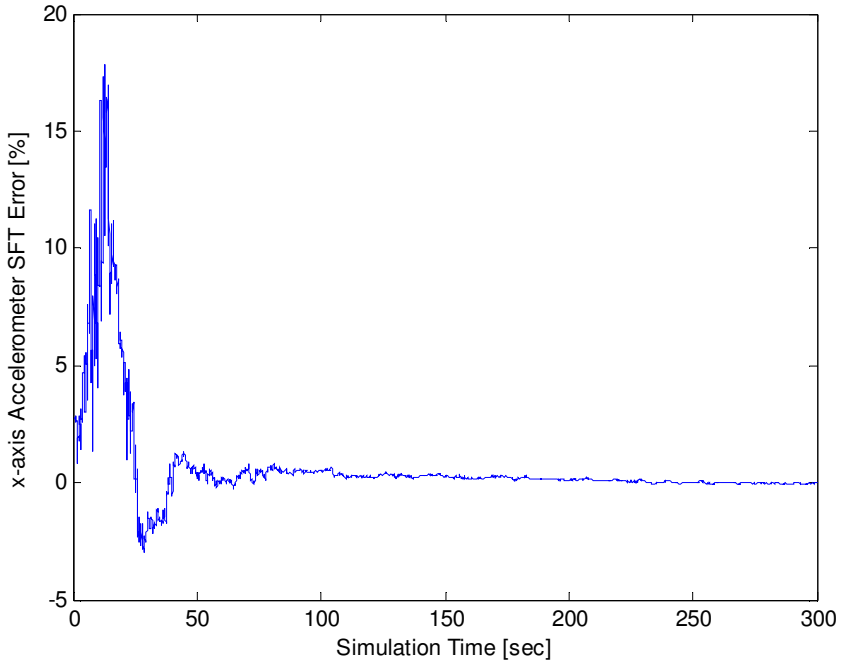

Figure 40. x-axis accelerometer SFT percentage error for 3-D case

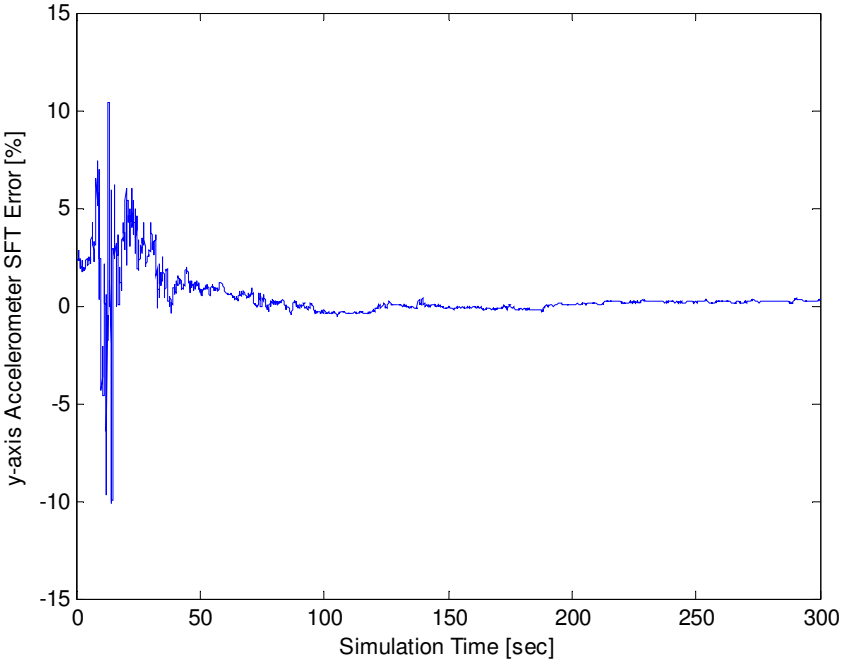

Figure 41. y-axis accelerometer SFT percentage error for 3-D case

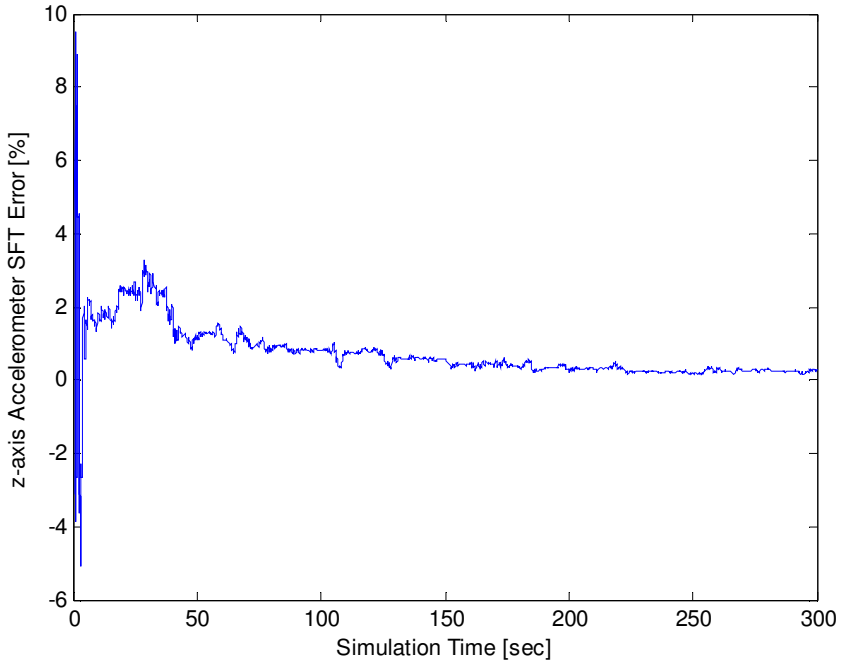

Figure 42. z-axis accelerometer SFT percentage error

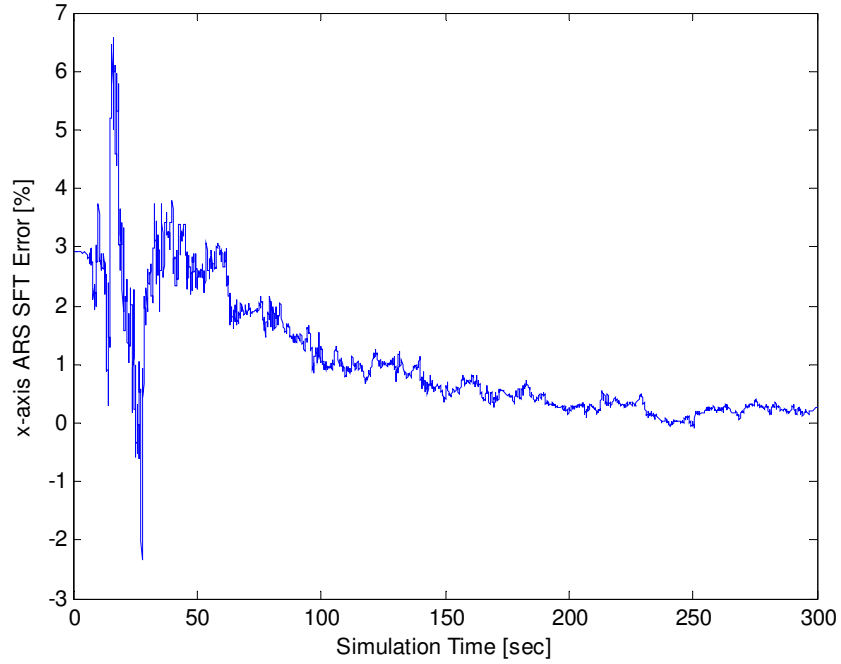

Figure 43. x-axis ARS SFT percentage error

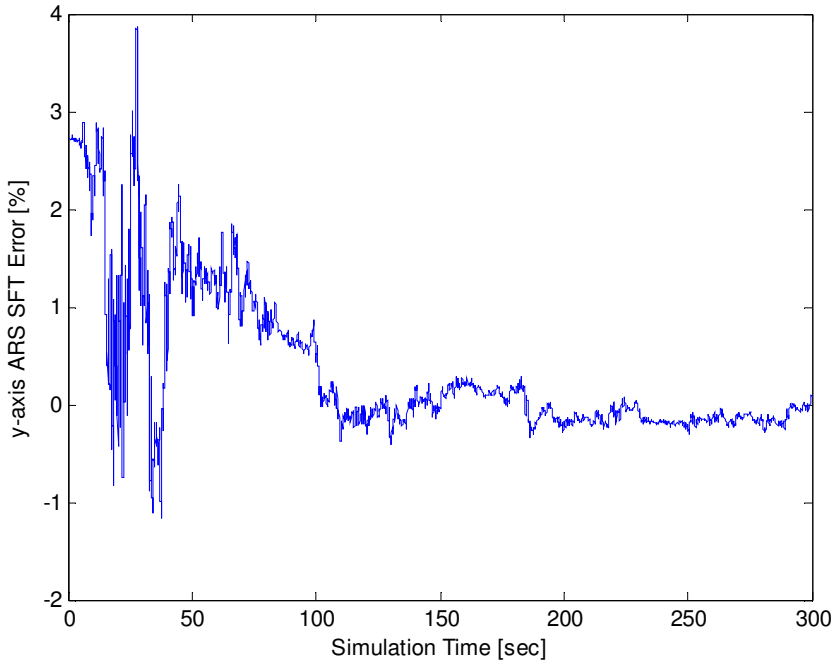

Figure 44. y-axis ARS SFT percentage error

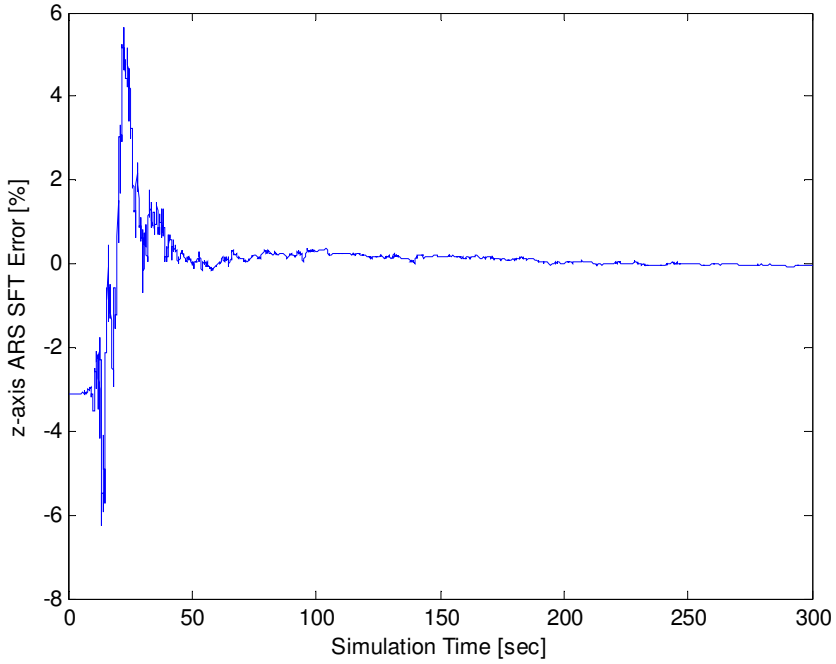

Figure 45. z-axis ARS SFT percentage error for 3-D case

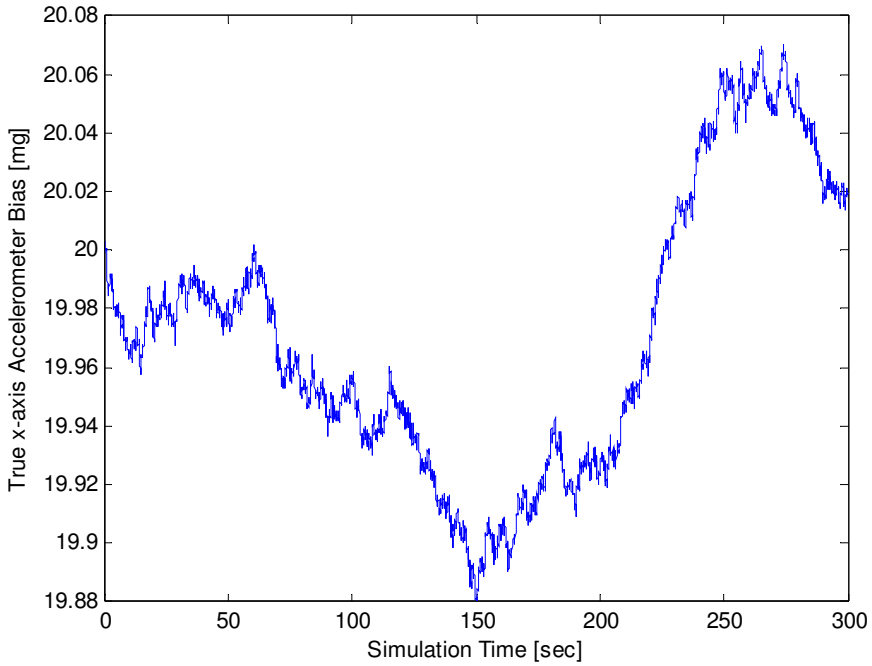

Figure 46. Random walk of x-axis accelerometer bias for 3-D case

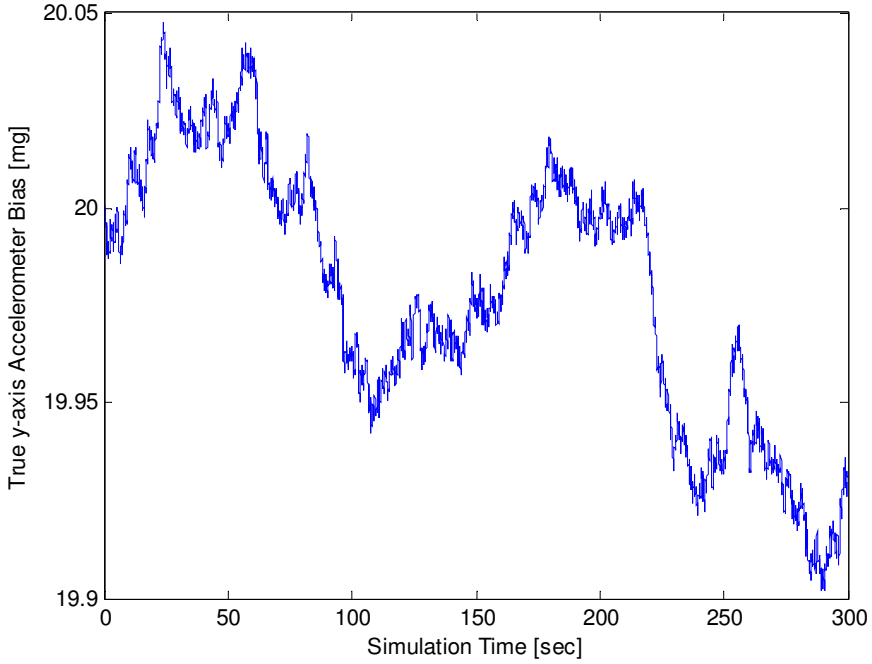

Figure 47. Random walk of y-axis accelerometer bias for 3-D case

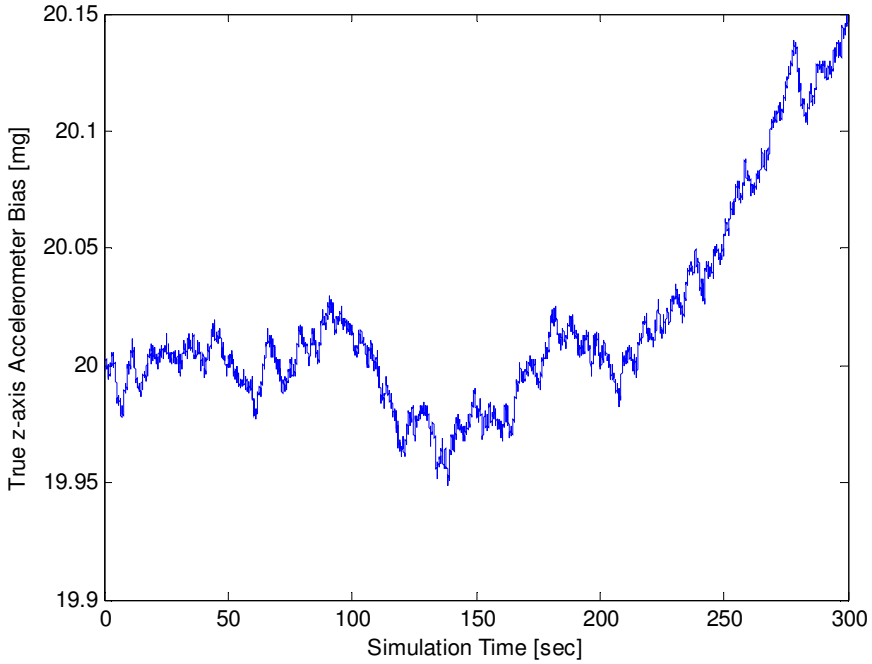

Figure 48. Random walk of z-axis accelerometer bias

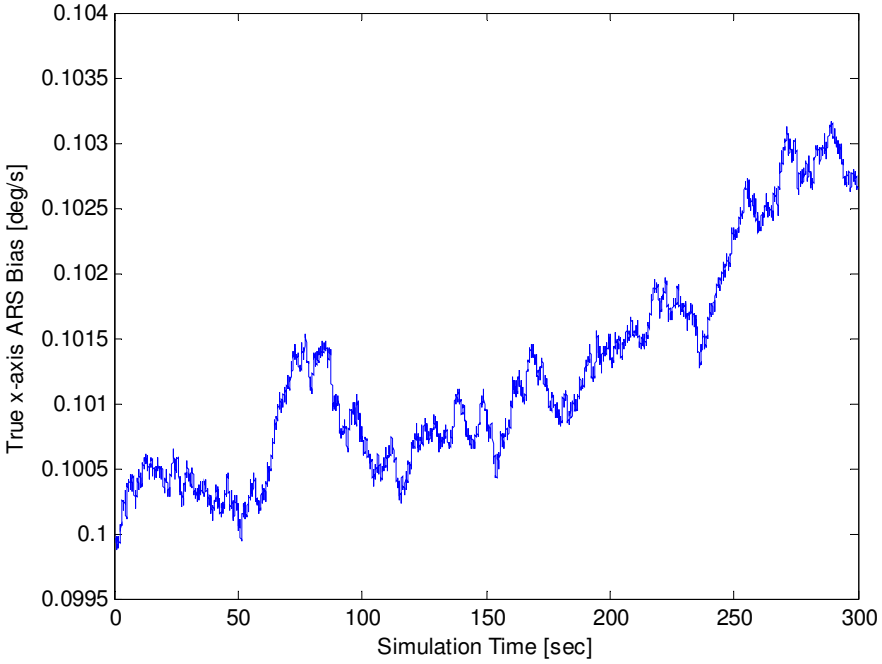

Figure 49. Random walk of x-axis ARS bias

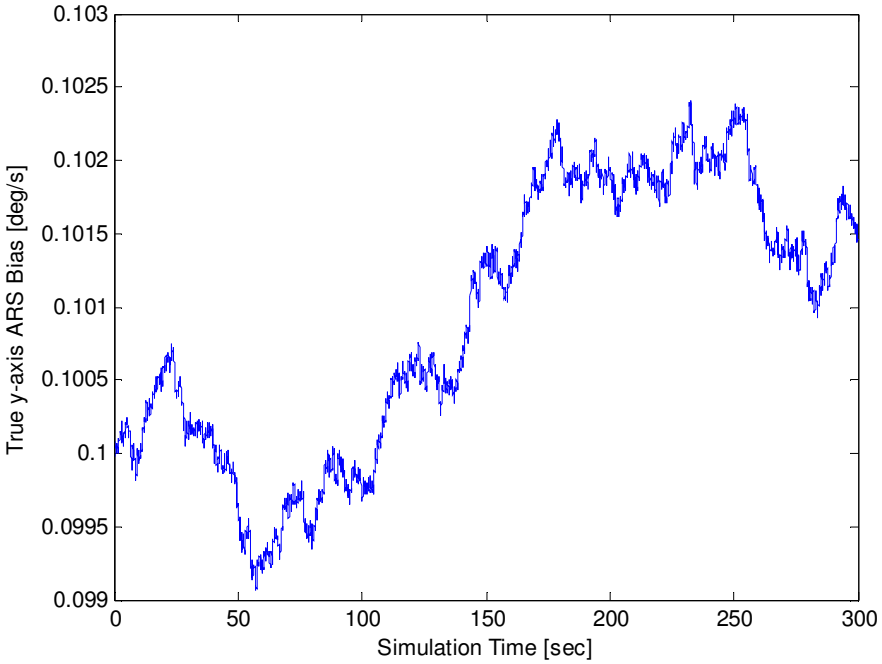

Figure 50. Random walk of y-axis ARS bias

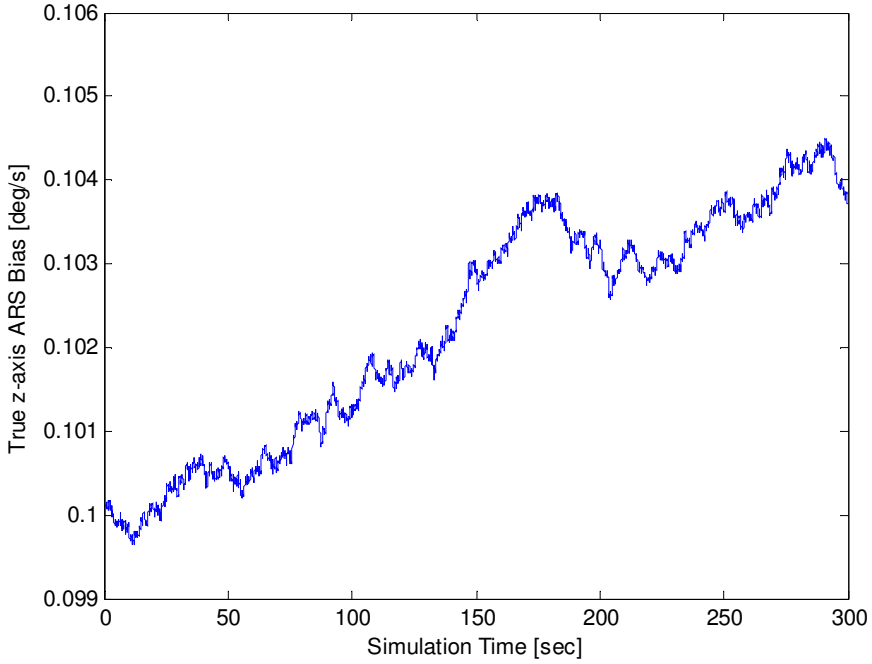

Figure 51. Random walk of z-axis ARS bias for 3-D case

Table 7. Statistical data for truth and state estimations for 3-D case

SD of North, East and Down position measurement errors	$1m$
SD of North, East and Down position estimation errors	$[0.17m \quad 0.22m \quad 0.15m]$
SD of North, East and Down velocity estimation errors	$[0.06m/s \quad 0.09m/s \quad 0.04m/s]$
SD of Yaw estimation error	$0.14°$
SD of Pitch estimation error	$0.05°$
SD of Roll estimation error	$0.07°$
SD of x,y,z-axes accelerometer noises	$5mg$
SD of x,y,z-axes accelerometers bias-rate noises	$0.0001g/s$
True initial value of x,y,z-axes accelerometer biases	$20mg$
SD of x,y,z-axes ARS noises	$0.05°/s$
SD of x,y,z-axes ARS bias-rate noises	$0.002°/s^2$
True initial value x,y,z-axes ARS biases	$0.1°/s$

5. REAL-TIME IMPLEMENTATION ON THE AGV

The majority of the code written in MatLab was for producing simulated measurements that were used by the KF. In reality, the measurements are produced by the GPS and IMU as the truck interfaces with the world. Therefore, the sections of the MatLab code that are required only for the simulation were not converted to C++. The other sections (KF and integration tool) were converted to C++ but required additional code to communicate with the hardware. Caleb Wells and Justin Bozalina (the computer scientists on the project) were of great help with communicating with the hardware and writing the C++ code. Appendix A contains the MatLab code, Appendix B contains the C++ code, Appendix C provides the global mapping transformations and Appendix D provides the necessary steps to start the INS and conduct testing.

Once the KF/IMU/GPS system was integrated, the tuning process began. The North, East and Down (NED) position of the AGV is measured by GPS (once the latitude, longitude and height above sea level have been converted). The KF solution for the NED position is therefore more robust than for the other states, which are not directly measured. The most difficult parameters to converge (through multiple updates from KF) were the biases and SFT's of the IMU because they are not propagated by the Runge-Kutta differential equation solver and therefore are only adjusted once the KF algorithm runs every 0.05 seconds. During GPS signal loss, these estimates do not change. They also directly affect the accuracy of the Euler angles.

It is very difficult to validate the KF estimates because the truth is not known. The most accurate measurements available are being used as inputs to the KF. Therefore, there is no reference by which the accuracy of the KF estimates can be measured. However, the precision/repeatability of the estimates can be demonstrated. Therefore, three tests were used to demonstrate the accuracy of the KF information.

The first test demonstrates the repeatability of the NED position estimate. The second shows the precision of the Yaw estimate. The third test included the mapping of an object onto the global map using the KF information and mapping transformations of Appendix C. In addition to these tests, there will also be intentional GPS blockages to show the drift of the model's solution. Also, the convergence of the biases and SFT's will be shown for completeness.

There is initialization required for each of these tests. The first step is to identify the origin of the ICS. The AGV will always start from this position for all the tests. Before the vehicle starts its run, the location of the tires is marked on the runway with the origin as the location of the IMU. The AGV will return to this position at the end of all tests.

5.1 Repeatability of NED position estimate

In order to test the repeatability of the NED position estimate from the KF, the vehicle returns to the origin where the precision error is calculated. The AGV returned to the origin within 11 cm along the North axis, 19 cm along the East axis and 8 cm along the Down axis. This test was performed many times with consistent results. The AGV initially traveled North on runway 35R and then turned clockwise. The vehicle then traveled northeast approaching the flight lab and returned along the same path, now traveling southwest. The AGV then traveled north for approximately 600 meters followed by a u-turn to travel south. The vehicle finished the test by approaching the origin from the South. Figure 52 shows the path of the AGV on the North-East plane.

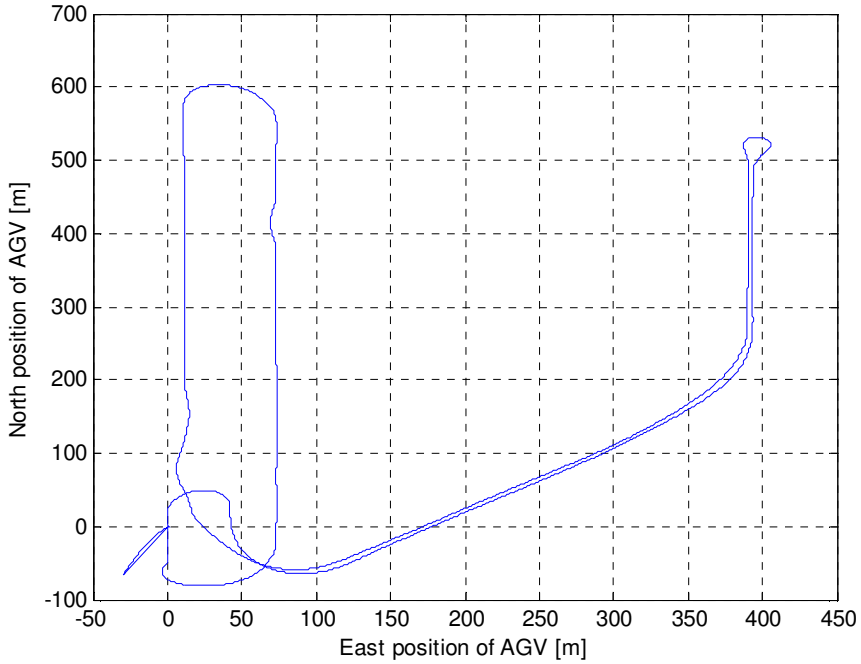

Figure 52. Path of AGV on the North-East plane

Once the AGV returned to the origin, the GPS signal was intentionally blocked for 61 seconds. The NED position estimates drifted due to the integration of uncompensated biases in the IMU. The NED position estimates drifted 66 meters, 30 meters and 9 meters over 61 seconds, respectively, indicating uncompensated biases in the x, y and z axis accelerometers of 3.6 mg, 1.6 mg and 0.5 mg. The yaw, pitch and roll angles drifted by 0.2°, 0.9° and 0.01°, respectively. If the GPS signal was blocked for 10 seconds, the drift in the NED position estimates would have been 1.8 meters, 0.8 meters and 0.25 meters, respectively.

5.2 Accuracy of the yaw angle estimate

Determining the accuracy of the yaw angle was one of the most difficult tasks in tuning the KF. The yaw angle is one of the most important states used by the controller and mapping software. To test the validity of the estimate, it has to be compared with the truth. At the Riverside campus, runway 35R has painted traffic lines that are parallel to North and South within 0.1°.

The offset in the runway was determined by driving the AGV parallel to these traffic lines and logging the GPS position information. After 600 meters, the East position drifted by ~ 1 meter indicating an overall offset of 0.1° clockwise about the D axis. This is the best reference available for determining the error in the yaw angle estimate.

The path of the AGV was shown in figure 52. The best yaw angle estimates were available at the end of the run and were taken while driving North and South. While driving north, the yaw angle estimate was between 359.2° and 359.9°. While driving south, the yaw angle estimate was between 179.4° and 179.8°. The plot of the yaw angle estimate for this section of the run is in figure 53.

The yaw angle estimates above indicate that there is an offset between the body-fixed x-axis of the IMU and the axis of the AGV that is parallel to the wheelbase and pointing forward. The Euler angle estimates provided by the KF are with respect to the IMU body-fixed co-ordinate system. It is impossible to mount the IMU such that all offsets are eliminated; rather, they are included when the state information is provided to the mapping and path control software. The exact offsets are not known but can be estimated (though not very well because the true orientation of the AGV is not known). For the yaw, it is estimated that the offset is 0.4°

meaning that the x-axis of the IMU is pointing 0.4° counter-clockwise from the centerline of the AGV.

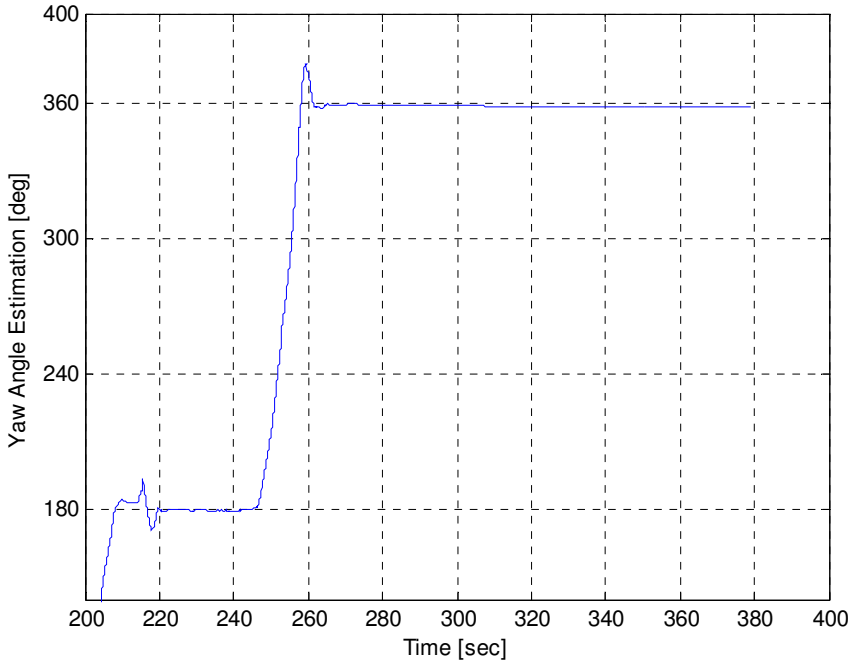

Figure 53. Yaw angle estimate of AGV during North to South and South to North operation

5.3 Precision of mapping solution

The final test was the most comprehensive. If multiple observations of an object are mapped within close proximity to one another (and to the correct location) then the information provided by the KF meets the performance requirements associated with obstacle mapping of the AGV project. The object mapped was a box measuring four meters by two meters and two meters in height.

For the final test, the AGV again started at the origin and was driven North for 200 meters and then performed a u-turn to drive South for an additional 400 meters. The AGV then performed another u-turn to approach the origin where it was stopped 10 meters to the East of the origin. The box was then rolled over the origin to make it easier to see its mapped location. The AGV then continued to drive North and performed another u-turn to head South. The vehicle then performed one last u-turn before the mapping process began, wherein the AGV was now approaching the box from the South.

Once the AGV was within 50 meters of the box, the time stamped returns from the SICK were logged. The SICK LMS [6] had a fixed sweep angle of -7° in order that the ground was contacted at ~ 20 meters. The box was mapped multiple times with the precision of these multiple observations being one meter (open the "mapping.avi" movie file in Appendix E to view the video demonstrating the mapping of this box).

5.4 State/Parameter estimates

There are an additional 18 figures included in this section. Figures 54 through 56 show the yaw, pitch and roll angle estimates. Figures 57 through 59 show the bias estimates for the x, y and z-axis accelerometers, respectively. Figures 60 through 62 show the bias estimates for the x, y and z-axis ARS's, respectively. Figures 63 through 65 display the SFT estimates for the x, y and z-axis accelerometers, respectively. Figures 66 through 68 display the SFT estimates for the x, y and z-axis ARS's, respectively.

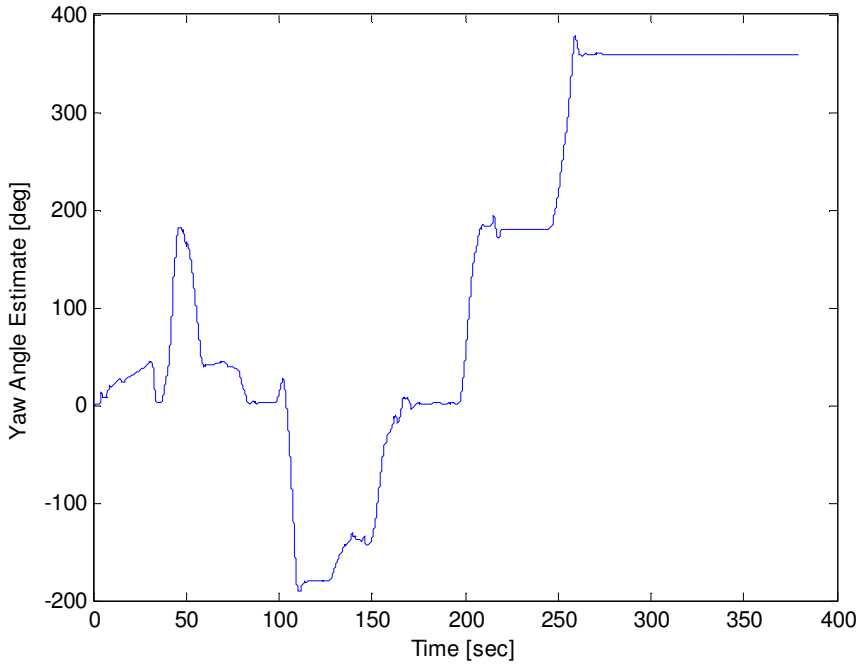

Figure 54. Yaw angle estimate

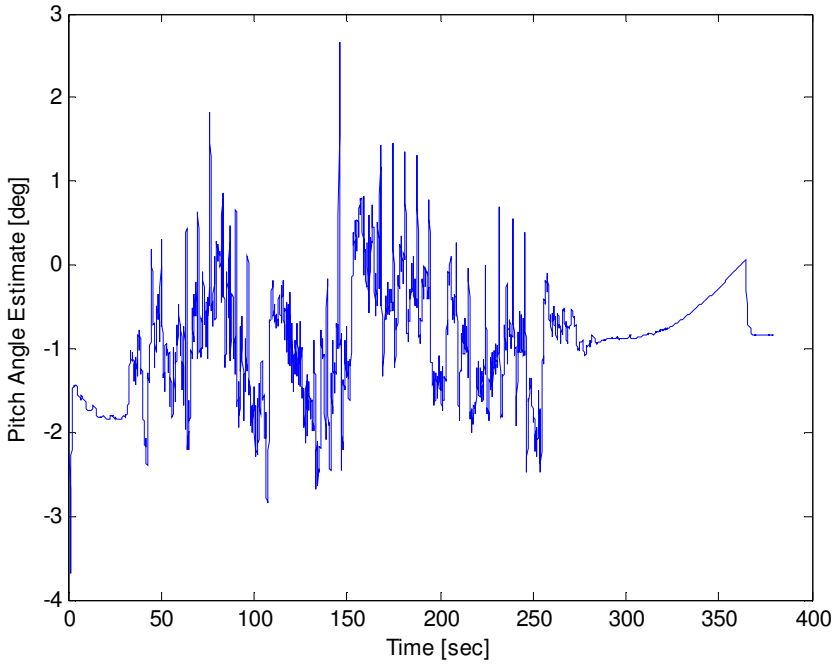

Figure 55. Pitch angle estimate

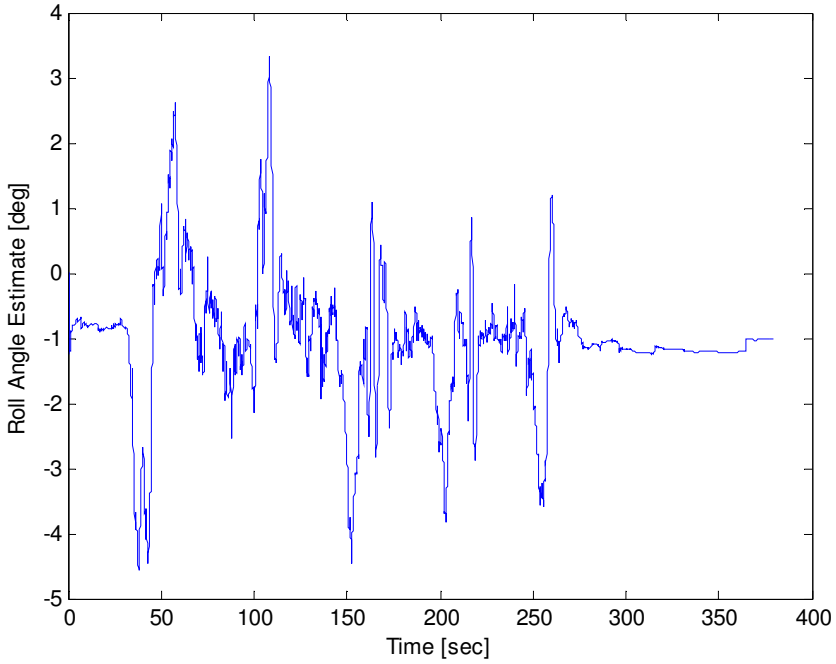

Figure 56. Roll angle estimate

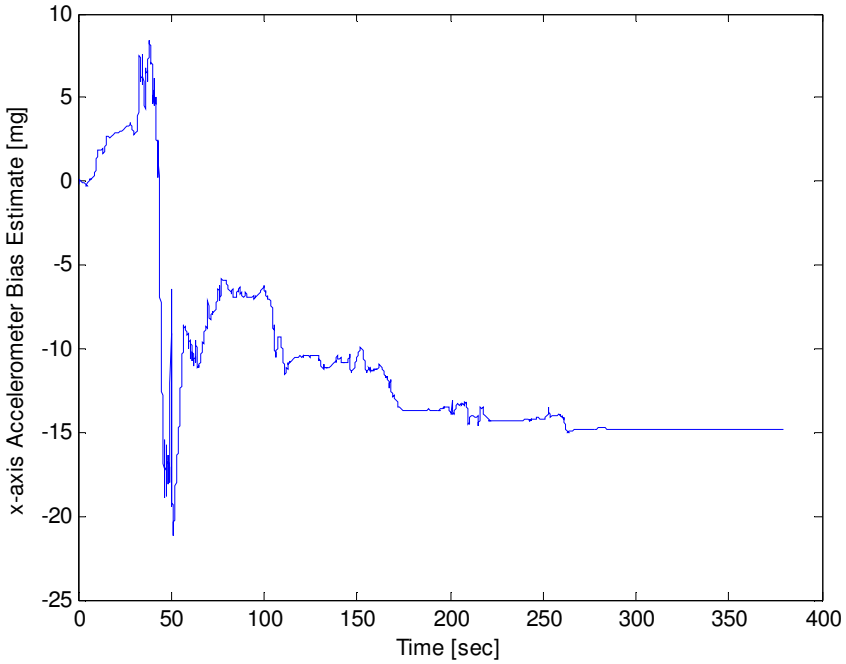

Figure 57. x-axis accelerometer bias estimate

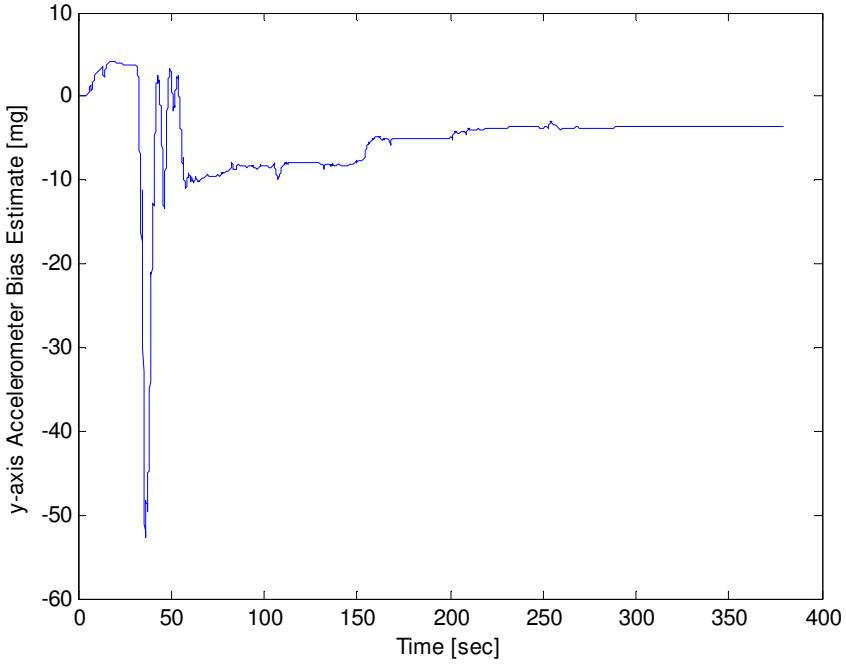

Figure 58. y-axis accelerometer bias estimate

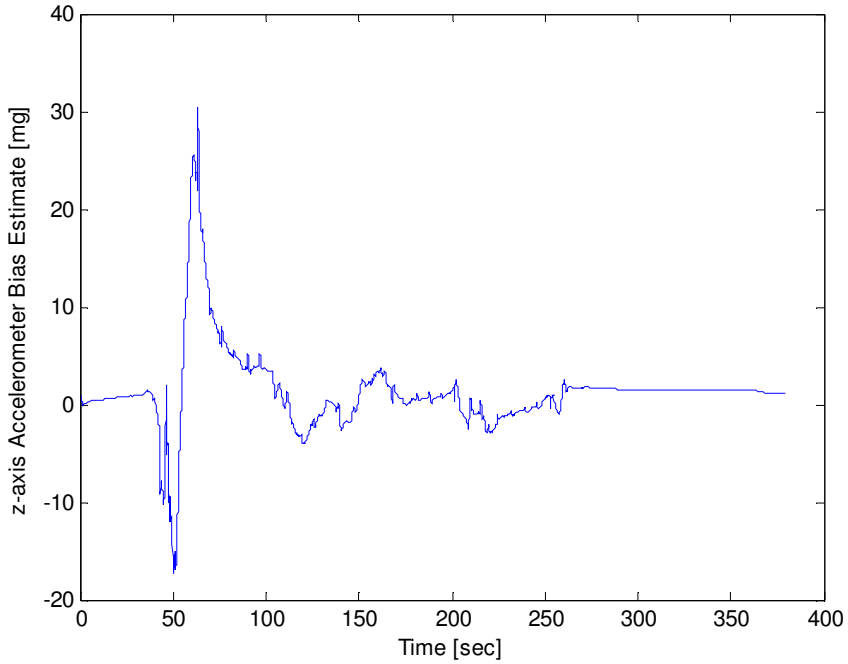

Figure 59. z-axis accelerometer bias estimate

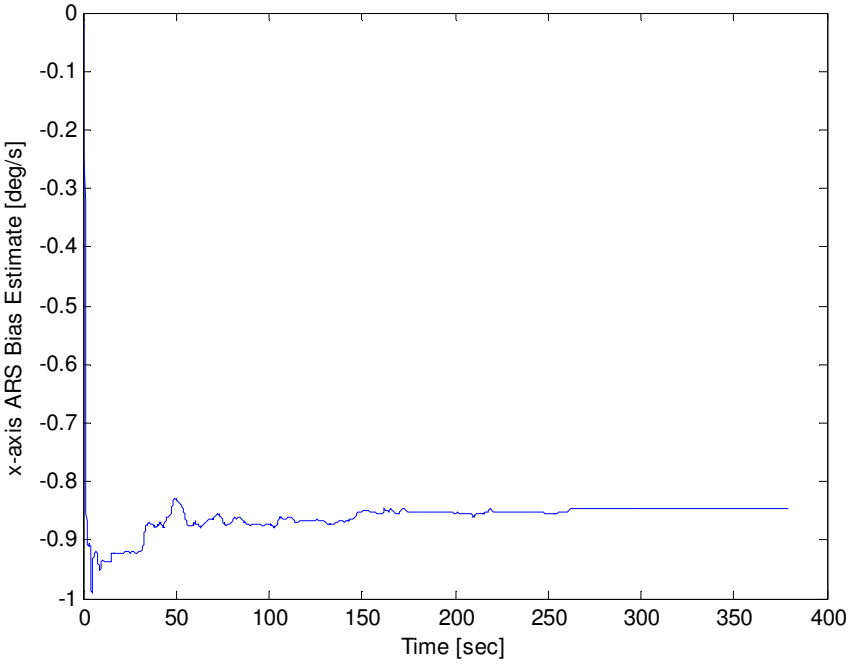

Figure 60. x-axis ARS bias estimate

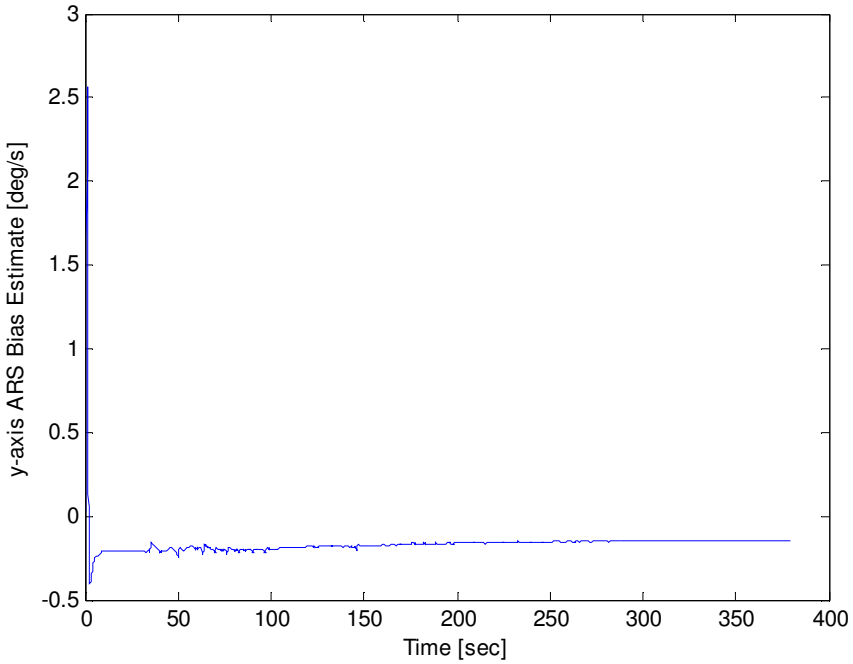

Figure 61. y-axis ARS bias estimate

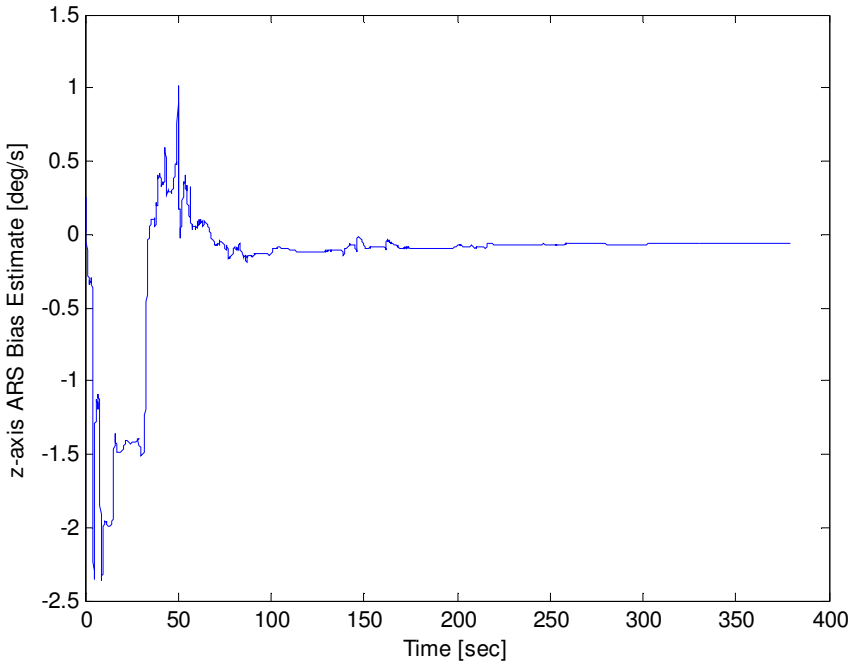

Figure 62. z-axis ARS bias estimate

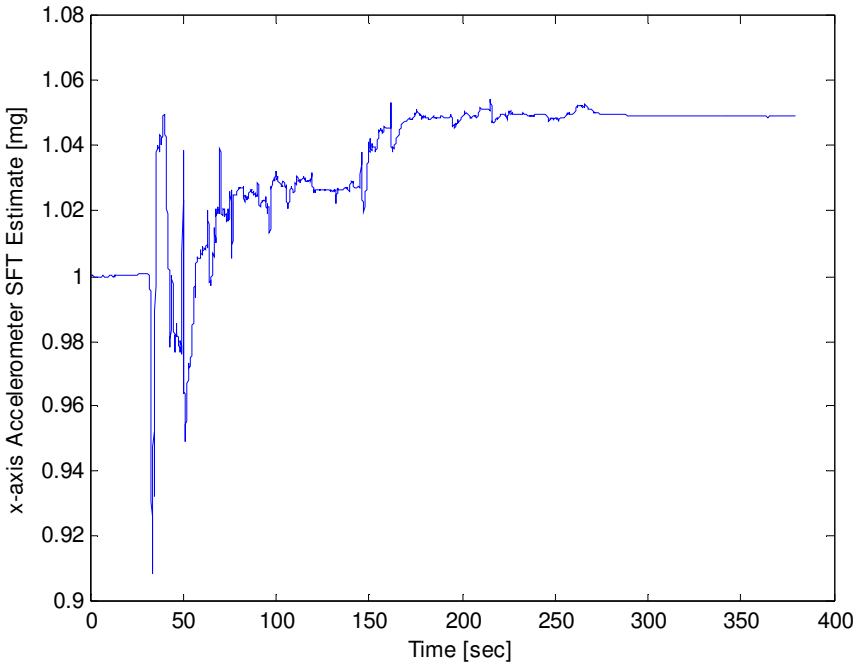

Figure 63. x-axis accelerometer SFT estimate

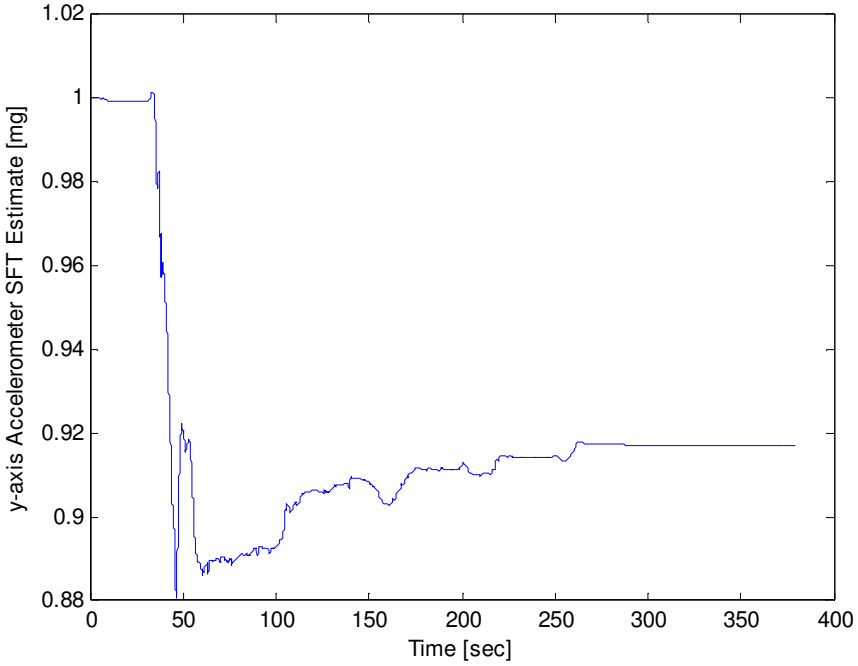

Figure 64. y-axis accelerometer SFT estimate

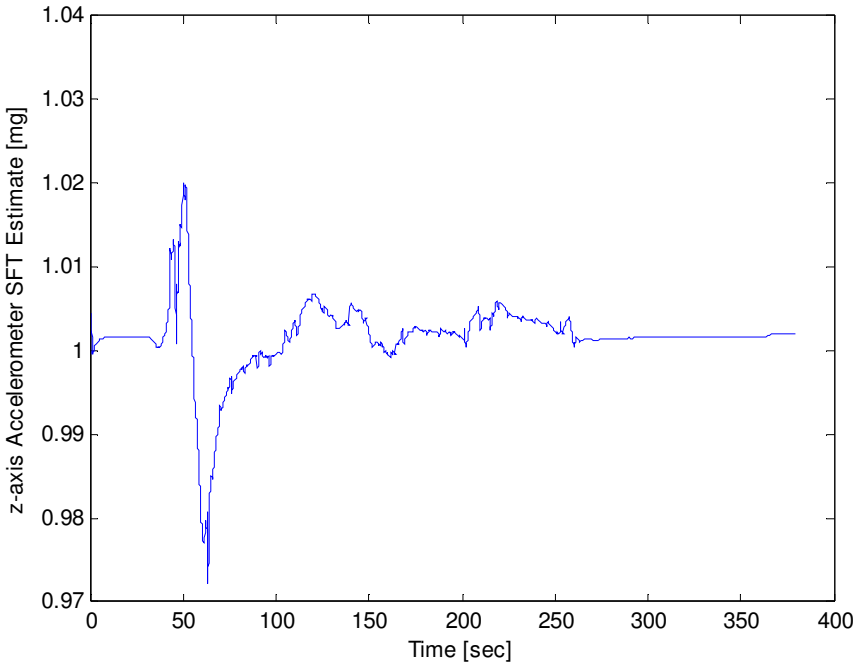

Figure 65. z-axis accelerometer SFT estimate

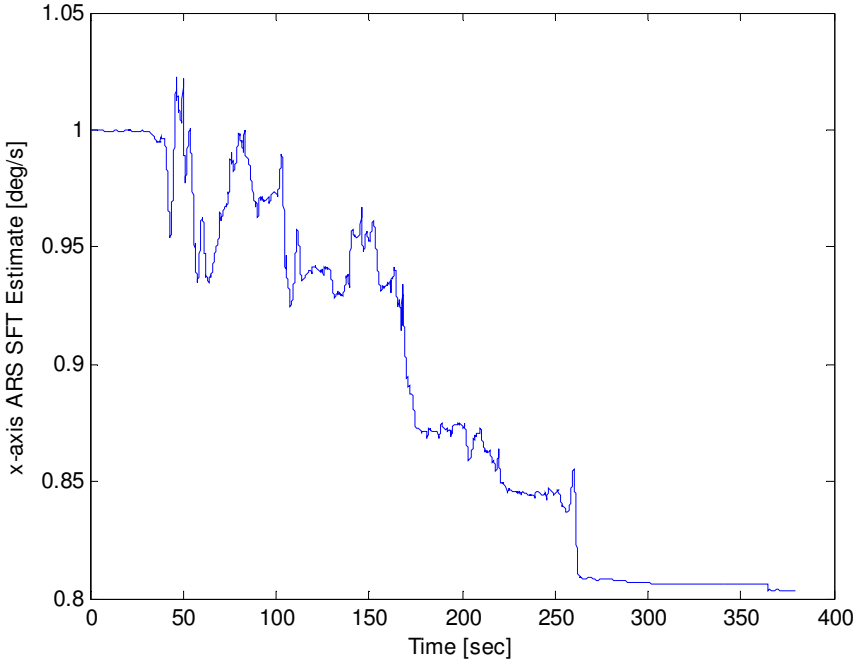

Figure 66. x-axis ARS SFT estimate

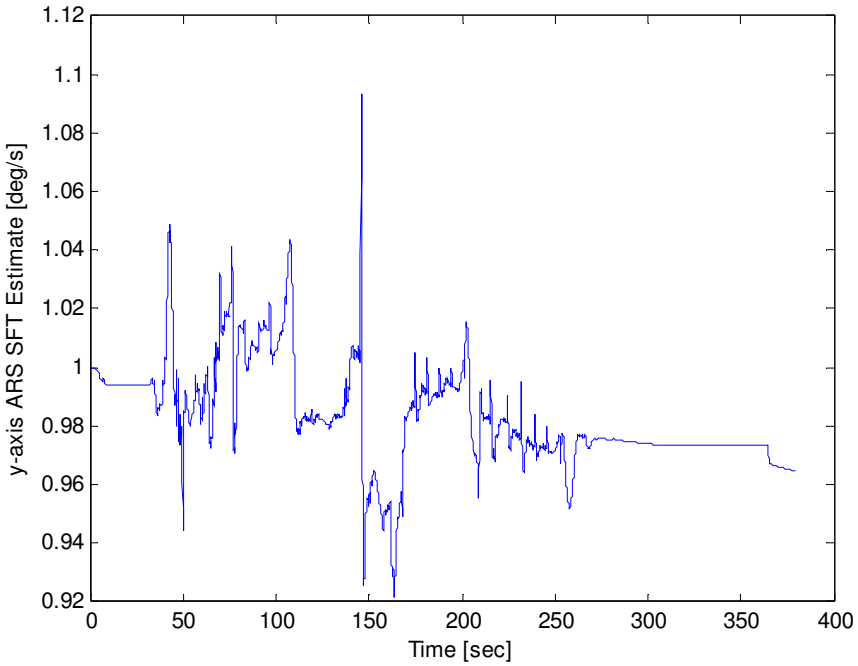

Figure 67. y-axis ARS SFT estimate

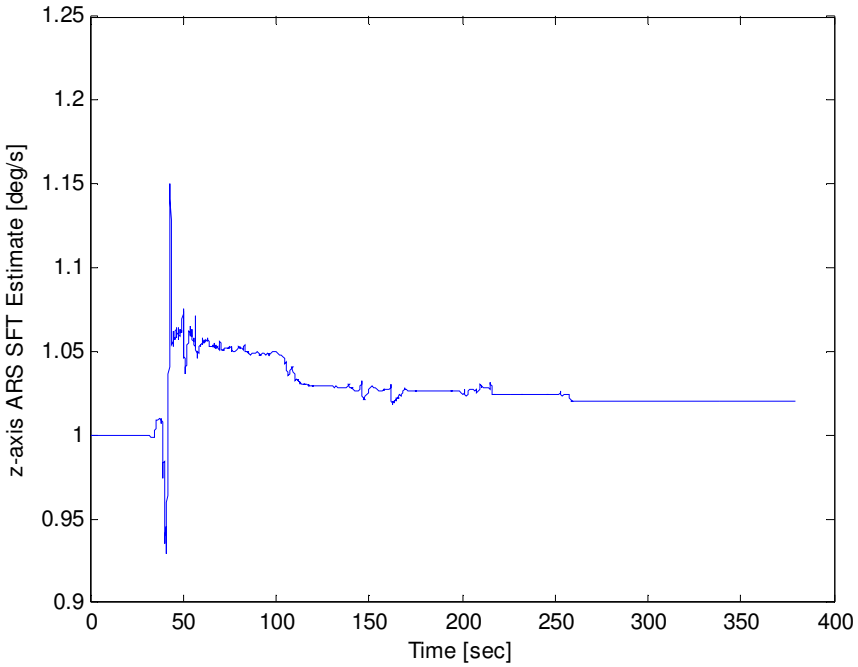

Figure 68. z-axis ARS SFT estimate

5.5 Tuning parameters comparison between real case and 3-D simulation case

The diagonal elements of the real P_0 matrix had values much higher than the diagonal elements of the P_0 matrix for the 3-D simulation (as can be seen in section 4.3). The same is true for the diagonal elements of $Q(t)$. The higher values indicate that there is more uncertainty in the IMU measurements for the real case. There is less certainty in the IMU measurements for multiple reasons. One reason is that the ideal noise characteristics used in the simulation are not present in the real measurements. Another reason for the discrepancy is that the biases were more active in the real IMU, which makes it more difficult for the KF to converge. Also, the real IMU measurements were corrupted with more noise than what the simulation included. However, the simulations still provided insight into how the KF reacts to the tuning parameters. Without this knowledge, it would have been much more difficult to arrive at the converged solution. The P_0, R_k and $Q(t)$ matrices are defined in equations (5.1) through (5.3) below.

Initial Estimation Error Covariance Matrix

$$\begin{aligned} P_0(\text{diag}(1-3,7-9)) &= 5 \\ P_0(\text{diag}(4-6)) &= 2 \\ P_0(\text{diag}(10-15)) &= 1 \\ P_0(\text{diag}(16-21)) &= 0.05 \end{aligned} \tag{5.1}$$

Measurement Noise Covariance Matrix

$$R_k = \begin{bmatrix} .1 & 0 & 0 \\ 0 & .1 & 0 \\ 0 & 0 & .2 \end{bmatrix} \tag{5.2}$$

Process Noise Covariance Matrix

$$\begin{aligned} Q(t) &= \text{diag}\begin{bmatrix} w_{Acc,x}^2 & w_{Acc,y}^2 & w_{Acc,z}^2 & w_p^2 & w_q^2 & w_r^2 \end{bmatrix} \\ w_{Acc,x}^2 &= w_{Acc,y}^2 = w_{Acc,z}^2 = (0.01g)^2 \\ w_p^2 &= w_q^2 = (0.5\,{}^{\circ}\!/_{s})^2,\ w_r^2 = (1.0\,{}^{\circ}\!/_{s})^2 \end{aligned} \tag{5.3}$$

5.6 Comparison of results with performance requirements

The first performance requirement of the navigation system was to provide six Hz updates of state information to the path controller, to the data frequency required to maintain a maximum stable vehicle speed of 40+ mph. The update rate provided by the navigation system is currently 71 Hz which is the update rate of IMU measurements to the KF. Therefore, the navigation system has satisfied the first performance requirement.

The second performance requirement of the navigation system was to provide a NED position error of less than 1/3 meter and yaw angular position error less than 0.5°. The NED position solution from the KF follows the differential GPS position closely because it is the most accurate position measurement available. The horizontal error of the NED position solution was 11 cm along the North axis and 19 cm along the East axis which is better than the 33 cm performance requirement. Also, the average yaw angle estimation error (excluding the offset) was 0.4° even though it was impossible to maintain a perfectly straight path along the traffic lines on runway 35R. Therefore, the second performance requirement has been satisfied.

The third performance requirement of the navigation system was that the IMU must have accelerometers and ARS's capable of sensing ± 10 g and $\pm$ 200°/s, respectively. The Watson Industries BA-604 IMU does not meet these requirements. This IMU is capable of ± 2 g and ± 100°/s. However, the Ring Laser Gyroscope (RLG) IMU discussed later will be capable of satisfying these requirements.

The fourth performance requirement of the navigation system was that it must include a KF that is capable of fusing the IMU/GPS information. A KF is required to provide all the necessary state information (NED position/velocity as well as orientation) which is mostly not available from GPS or IMU alone. Since the major contribution of this thesis was the development of the KF, this performance requirement is satisfied.

The fifth performance requirement of the navigation system was that the drift rate of the Euler angles be less than 30°/hr. When there is a momentary loss in the GPS signal, the navigation solution provided by the KF will drift. As was shown earlier, after one minute the drift in the Euler angles can be between 0.01° and 0.9°. Since the yaw angle is used as the benchmark, the drift rate of the navigation solution was 12°/hr. Therefore, the fifth performance requirement has been satisfied.

The next two performance requirements were that the differential GPS must be available in the Mojave Desert and that the communication from C++ to the hardware must be serial or TCP/IP. The differential GPS signal used by the navigation system is provided by OmniSTAR and is available in the Mojave Desert. The communication between the hardware and C++ was serial. Therefore, the sixth and seventh performance requirements have been satisfied.

The last performance requirement of the navigation system was that the hardware must have shock survivability greater than 20 g for 5 ms. This information has not been provided by Watson Industries, therefore it is unknown if it meets the requirements. However, during the operation of the AGV, the hardware did survive the vibrations. Nonetheless, the last performance requirement has not been satisfied.

The current navigation system does not meet all the performance requirements. However, with the replacement of the current IMU with the IMU in the SPAN # 2 package (Honeywell G2-H58) these requirements will be satisfied. However, the failed performance requirements did not hinder the overall performance of the INS. The accelerations and angular rates sensed by the IMU were never above its maximum range. Also, the IMU survived the vibrations it experienced during the operation of the AGV on the runways at Riverside. However, the runway environment does not fully define the strength of vibrations that the AGV can experience in an off-road or less smooth environment.

6. CONCLUSIONS

The current navigation system is sufficient for following waypoints, as was the previous INS. However, the current INS also can withstand a loss in GPS signal for no more than 10 seconds where the NED position estimates will drift by at most two meters. With the orientation information provided by the current INS, a locally 'global' map of the terrain is possible which is the first step toward identifying obstacles that the AGV must avoid.

The current INS does not satisfy all the performance requirements developed through the design analysis. However, all of the performance requirements would be satisfied by the integration of the Honeywell G2-H58 RLG IMU into the INS. The accuracy of the Euler angles and the drift associated with the biases of the three accelerometers and three ARS's will be improved significantly. With the biases more accurately estimated, the INS could withstand a longer GPS signal loss.

The simulations developed in this thesis helped tremendously in learning the behavior of the KF, especially when adjusting the tuning parameters. Also, the simulations show the observability of each of the states/parameters (whether or not they can be estimated). Without these simulations, it would have been much more difficult with respect to time and effort to develop an INS that performed as well as the current INS.

The KF is a very useful tool for navigation. Without the KF, the biases of the IMU sensors would not be estimated. Integrating the IMU measurements with these biases would introduce integration errors that would render the navigation solution useless. If the biases were fully known, there is still noise in the measurements that affect the accuracy of the solution. The KF essentially provides the most accurate solution possible.

7. SUMMARY

The design analysis was a crucial step in this project. It was intentionally placed in the beginning of this thesis because it should be the first step in solving a problem. The needs statement focuses on posing the problem correctly, in order that any ambiguity is removed. The functional/performance requirements present the necessary requirements a candidate solution must possess, and how well it must perform them. Much time is spent analyzing the problem to determine the levels of performance required of the design.

Once the framework for the solution was developed, some background on the KF was presented. The KF is a tool which has many applications. For navigation, it can take body-fixed accelerations and angular-rates, along with an inertial linear position update and provide linear positions/velocities as well as the yaw, pitch and roll angles of the AGV. If there were no data corruption by noise, biases and/or SFT's, a KF would not be required. The measurement data could be simply integrated and arrive at the true solution. However, all measurements are corrupted and thus the use of a KF is warranted.

The simulations developed in section four were used to confirm that the KF could indeed provide a viable navigation solution. Even though the truth was available, it was corrupted before it was sent to the KF, mimicking a real measurement. Since the truth was known, the solution provided by the KF could be evaluated, a luxury not available in reality.

Appendix C described the obstacle mapping process in detail and provided definitions of the Euler angles (yaw, pitch and roll). It is very important that every system on the AGV that requires orientation uses the correct order and definitions of the Euler angles. Along with the orientation definitions, the vector analysis for mapping a SICK return onto the global map is also critical to the operation of the AGV. If the mapping software does not correctly map the environment, the AGV cannot determine the safest route to the goal.

Although simulation is a very powerful and useful tool, the ultimate test of any system is how it performs in its operating environment. Without the truth as a gauge, it was difficult to determine the tests required to show that the current INS performs adequately. However, the results of the tests employed show that the KF algorithm developed in this thesis is useful for navigation of an AGV.

8. RECOMMENDATIONS FOR FURTHER STUDY/DEVELOPMENT

It is recommended that an IMU with the lowest bias drift/walk be purchased that is within the current budget. The navigation system on the AGV will benefit from an IMU that has a bias drift/random walk less than the current IMU. During GPS outages, the information from the IMU (linear accelerations and angular rates) is integrated to provide the linear positions/velocities as well as the Euler angles. When the KF is active (GPS is available) it is compensating for the biases in the IMU, but the biases can never be fully compensated for because they drift and walk. However, the amount that they drift/walk is slow relative to the bandwidth of the KF. The problem arises when the KF is not able to compensate for them which occurs when the GPS signal is unavailable.

There is not an IMU available that eliminates the problem of biases. A better IMU will only provide a viable, uncorrected solution longer than the current IMU. A viable solution is one in which the errors developed in the linear positions and Euler angles are acceptable. The lower the bias drift/walk, the longer an acceptable navigation solution can be used by the AGV in the absence of GPS.

With the addition of a new IMU, there will need to be additional development. Outside of communication and mounting issues, there are additional tuning requirements. Since the new IMU will indeed provide a more accurate solution, the tuning parameters described earlier will have to be adjusted. The recommended process is to start with the current tuning parameters and then lower the diagonals of the P_0 matrix that pertain to the biases and SFT's of the IMU. Also, repeat this process for the diagonals of the $Q(t)$ matrix. Lowering the values of the elements of these matrices will place more weight on the IMU.

REFERENCES

[1] Defense Advanced Research Projects Agency, DARPA and the DARPA Grand Challenge. http://www.darpa.mil/grandchallenge. Accessed: January 2005.

[2] NovAtel, NovAtel SPAN Package. http://www.novatel.com/products/span.htm. Accessed: June 2005.

[3] Andrews, A.P. and Grewal, M.S. and Weill, L.R., 2001, *Global Positioning Systems, Inertial Navigation, and Integration*, John Wiley & Sons, New York, NY.

[4] Crassidis, J.L. and Junkins, J.L., 2004, *Optimal Estimation of Dynamic Systems*, Chapman and Hall, New York, NY.

[5] Watson Industries, Watson Industries IMU BA-604. http://www.watson-gyro.com. Accessed: August 2005.

[6] SICK Automatic Identification Sensors, SICK LMS221-30206. http://www.sickusa.com. Accessed: January 2005.

APPENDIX A

MatLAB code for simulating 1-D IMU/GPS measurements:

```
% Author:  Craig Odom

% Simulated measurement code for 1-D case

clc;
clear;
close all;

global x_acc_eta Q offset amp w_length

g = 9.807;

% Noise parameters
GPS_N_noise = 1;
x_acc_noise = 0.005;
x_acc_eta = 0.0001;

% IMU--->GPS offsets
x_offset = -1.0;

offset = [x_offset];

% Scale factor actual and nominal (V/g, V/(deg/s))
S_act_x_acc = 2.57;

S_nom_x_acc = 0.4;

% Time parameters
t_begin = 0;
dt_cont = 0.01;
dt_disc = 0.05;
t_final = 200;
time = t_begin:dt_cont:t_final;
time_disc = t_begin:dt_disc:t_final;

% True motion
amp = 200;
w_length = 50;

N_pos = 0;
N_vel = 2*amp*pi/w_length;
x_acc_bias_true = 0.02;
```

```
x0_true_motion(1:3,1) = [N_pos N_vel x_acc_bias_true]';

N_acc = -4*amp*sin(2*pi*time/w_length)*pi^2/w_length^2;

% Calling the ode4 function (Runge Kutta 4th order)
x_true = ode4(@true_motion_1D,time,x0_true_motion);

N_pos = x_true(:,1);
N_vel = x_true(:,2);
x_acc_bias_true = x_true(:,3);

x_acc_body = N_acc/g;

% Scale factor errors
S_a_x = S_act_x_acc*S_nom_x_acc;

S_accel = [S_a_x];

% Building IMU measurements
for i=1:length(time)

   x_acc_meas(i) = S_accel*x_acc_body(i) + x_acc_bias_true(i) + x_acc_noise*randn;

end

% Rounding the IMU data for true resolution
x_acc_meas = round(x_acc_meas*1000)/1000;

% Building the GPS measurements
for i=1:length(time_disc)

   j = (dt_disc/dt_cont)*i - (dt_disc/dt_cont - 1);

   N_offset(i) = offset;

   GPS_N_meas(i,1) = N_pos(j) + N_offset(i) + GPS_N_noise*randn;

end

% Rounding the GPS data
GPS_N_meas = round(GPS_N_meas*100)/100;

% Building the y_tilde vector
for i=1:length(time_disc)

   y_tilde(i,1) = GPS_N_meas(i,1);

end
```

```
GPS_data = [GPS_N_meas];

IMU_data = [x_acc_meas'];

save Initial_1D.mat

run EKF_1D
```

MatLab code that runs Runge-Kutta 4th order to produce the true motion for the 1-D case:

```
% Author:  Craig Odom

% Simulated truth code for 1-D case

function f = true_motion_1D(t,x)

global x_acc_eta amp w_length

% Initializing the functions
f = zeros(3,1);

% Equations of motion
f(1) = x(2);
f(2) = -4*amp*sin(2*pi*t/w_length)*pi^2/w_length^2;
f(3) = x_acc_eta*randn;
```

MatLab code that runs the Kalman Filter algorithm for the 1-D case:

```
% Author:  Craig Odom

% Kalman Filter code that uses simulated measurements from Initial_1D.mat

clc;
clear;
close all;

load Initial_1D.mat

global x_acc_m Q offset

% Initial Estimation Error Covariance Matrix
for i=1:4

   if i<=2
```

```
        P0(i,i) = 0.5;

    elseif i==3

        P0(i,i) = 0.005;

    elseif i==4

        P0(i,i) = 0.005;

    end

end

% Process Noise Covariance Matrix
x_acc_noise_mss = x_acc_noise*g;

Q = [x_acc_noise_mss^2];

% Measurement Noise Covariance Matrix
R = [GPS_N_noise^2];

% Initial State Estimation
x_start = [0 25 0 1];

x_k_neg = x_start';

P_k_neg = P0;

step = dt_disc/dt_cont;

z = 1;

iteration = 0;

print = 0;

% Extended Kalman Filter
for i=1:length(time)

    % step builds until a new GPS measurement is available
    if step == dt_disc/dt_cont;

        h = x_k_neg(1,1) + offset;

        % Measurement Sensitivity Matrix
        H = [1 0 0 0];
```

```
    % Kalman Gain Matrix
    K = P_k_neg*H'*inv(H*P_k_neg*H' + R);

    Inn(:,i) = y_tilde(z,1) - h;

    % Correction to state vector
    x_k_pos = x_k_neg + K*(Inn(:,i));

    % Correction to P matrix
    P_k_pos = (eye(length(P0)) - K*H)*P_k_neg;

    step = 0;

    z = z + 1;

end

% Saving all state information
x_plot(:,i) = x_k_pos;

% Augmenting the IMU measurements from g's to m/s/s and deg/s to rad/s
x_acc_m = x_acc_meas(i)*g;

% Setting the first part of the initial condition vector for Runge
% Kutta
x0(1:length(P0),1) = x_k_pos;

% Converting elements of P matrix to an array
for d=1:length(P0)

    j = length(P0)*d - (length(P0)-1);
    k = length(P0)*d;

    P_k_pos_col(j:k,1) = P_k_pos(d,1:length(P0));

end

% Setting the last part of the initial condition vector for Runge Kutta
x0(length(P0)+1:length(P0)+length(P0)^2,1) = P_k_pos_col;

% Calling the ode4 (Runge Kutta 4th order) integration tool
x_est = ode4(@Prop_1D,[0 0.01],x0);

% Taking the last part of the RK solution
P_est = x_est(length(x_est(:,1)),length(P0)+1:length(P0)+length(P0)^2);

% Converting the array P_est into the elements of P_k_neg
for d=1:length(P0)
```

```
        j = length(P0)*d - (length(P0)-1);
        k = length(P0)*d;
        P_k_neg(d,1:length(P0)) = P_est(j:k);

    end

    % Converting the first part of the solution to x_k_neg
    x_k_neg = x_est(length(x_est(:,1)),1:length(P0))';

    % x_k_pos/P_k_pos are the best estimates available
    x_k_pos = x_k_neg;

    P_k_pos = P_k_neg;

    step = step + 1;

    % Printing out the simulation time every second
    if iteration == 100

        print = print + 1;

        Sim_time = print

        iteration = 0;

    end

    iteration = iteration + 1;

end

% Solving for the percentage error of the SFT estimate
for i=1:length(time)

    x_acc_per_error(i) = (S_a_x - 1/x_plot(4,i))/S_a_x*100;

end

% Saving all the data
save EKF_1D.mat

% Printing all the figures
figure (1)
plot(time,x_plot(1,:) - N_pos')
xlabel('Simulation Time [sec]')
ylabel('North Position Estimation Error [m]')
axis([0 200 -2 2])
```

```
figure (2)
plot(time,x_plot(2,:) - N_vel')
xlabel('Simulation Time [sec]')
ylabel('North Velocity Estimation Error [m/s]')
axis([0 200 -1 1])

figure (3)
plot(time,(x_acc_bias_true - x_plot(3,:)'/9.807)*1000)
xlabel('Simulation Time [sec]')
ylabel('x-axis Accelerometer Bias Estimation Error [mg]')

figure (4)
plot(time,x_acc_per_error)
xlabel('Simulation Time [sec]')
ylabel('x-axis Accelerometer SFT Error [%]')

figure (5)
plot(time,x_acc_bias_true*1000)
xlabel('Simulation Time [sec]')
ylabel('True x-axis Accelerometer Bias [mg]')

figure (6)
plot(time,N_pos)
xlabel('Simulation Time [sec]')
ylabel('True North Position of AGV [m]')
```

MatLab code that runs Runge-Kutta 4th order to produce the *apriori* estimates (including the estimation error covariance matrix):

```
% Author:  Craig Odom

% Propagation code for apriori solution for 1-D case

function f = Prop_1D(t, x)

global x_acc_m Q

% Initializing the functions
f = zeros(4,1);

% Truth model
a_x = x(4)*(x_acc_m - x(3));

% Equations of motion
f(1) = x(2);
f(2) = a_x;
```

```
f(3) = 0;
f(4) = 0;

% Process Noise mapping matrix
G = [0
     x(4)
     0
     0];

% Partial derivatives matrix
F23 = -x(4);
F24 = x_acc_m - x(3);

F = [0 1 0 0
     0 0 F23 F24
     0 0 0 0
     0 0 0 0];

st = length(F) + 1;

% Filling the P matrix with the states
for i=1:length(F)

    for j=1:length(F)

        P(i,j) = x(st);

        st = st + 1;

    end

end

% Variation of Ricatti equation
P_dot = F*P + P*F' + G*Q*G';

st = length(F) + 1;

% Filling the functions with the elements of P_dot
for i=1:length(F)

    for j=1:length(F)

        f(st) = P_dot(i,j);

        st = st + 1;

    end
```

```
end
```

MatLAB code for simulating 2-D IMU/GPS measurements:

```
% Author:  Craig Odom

% Simulated measurement code for 2-D case

clc;
clear;
close all;

global x_acc_eta y_acc_eta omega_r_eta g Q offset amp w_length

g = 9.807;

% Noise parameters

GPS_N_noise = 1;
GPS_E_noise = 1;
x_acc_noise = 0.005;
y_acc_noise = 0.005;
omega_r_noise = 0.05;
x_acc_eta = 0.0001;
y_acc_eta = 0.0001;
omega_r_eta = 0.002;

% IMU--->GPS offsets

x_offset = -1.0;
y_offset = 0.2;

offset = [x_offset;y_offset];

% Scale factor actual and nominal (V/g, V/(deg/s))

S_act_x_acc = 2.57;
S_act_y_acc = 2.56;
S_act_omega_r = 0.097;

S_nom_x_acc = 0.4;
S_nom_y_acc = 0.4;
S_nom_omega_r = 10;

% Time parameters
```

```
t_begin = 0;
dt_cont = 0.01;
dt_disc = 0.05;
t_final = 200;
time = t_begin:dt_cont:t_final;
time_disc = t_begin:dt_disc:t_final;

% True motion

amp = 200;
w_length = 50;

N_pos = 0;
E_pos = 0;
N_vel = 2*amp*pi/w_length;
E_vel = 5;
yaw = 0;
x_acc_bias_true = 0.02;
y_acc_bias_true = 0.02;
omega_r_bias_true = 0.1;

x0_true_motion(1:7,1) = [N_pos E_pos N_vel E_vel x_acc_bias_true y_acc_bias_true
omega_r_bias_true]';

N_acc = -4*amp*sin(2*pi*time/w_length)*pi^2/w_length^2;

E_acc = zeros(1,length(time));

% Calling the ode4 function (Runge Kutta 4th order)
x_true = ode4(@true_motion_2D,time,x0_true_motion);

N_pos = x_true(:,1);
E_pos = x_true(:,2);
N_vel = x_true(:,3);
E_vel = x_true(:,4);
x_acc_bias_true = x_true(:,5);
y_acc_bias_true = x_true(:,6);
omega_r_bias_true = x_true(:,7);

% Building the true yaw motion
for i=1:length(time)-1

    yaw(i+1) = atan2(E_vel(i+1),N_vel(i+1));

    yaw_rate(i) = (E_acc(i)*N_vel(i) - N_acc(i)*E_vel(i))/(E_vel(i)^2 + N_vel(i)^2);

    if i==length(time)-1
```

```
      yaw_rate(i+1) = (E_acc(i+1)*N_vel(i+1) - N_acc(i+1)*E_vel(i+1))/(E_vel(i+1)^2 +
N_vel(i+1)^2);

   end

end

y = yaw;

% Building the true rotation
for i=1:length(time)

   omega_r(i) = yaw_rate(i);

end

% Direction Cosine Matrix
for i=1:length(time)

   DCM(1,1) = cos(y(i));
   DCM(1,2) = sin(y(i));
   DCM(2,1) = -sin(y(i));
   DCM(2,2) = cos(y(i));

   x_acc_body(i) = DCM(1,:)*[N_acc(i) E_acc(i)]';

   y_acc_body(i) = DCM(2,:)*[N_acc(i) E_acc(i)]';

end

x_acc_body = x_acc_body/g;
y_acc_body = y_acc_body/g;

omega_r = omega_r*180/pi;

acc_body = [x_acc_body;y_acc_body];

omega_body = [omega_r];

% Scale factor errors
S_a_x = S_act_x_acc*S_nom_x_acc;
S_a_y = S_act_y_acc*S_nom_y_acc;

S_g_r = S_act_omega_r*S_nom_omega_r;

S_accel = [S_a_x 0
          0 S_a_y];
```

```
S_gyro = [S_g_r];

% Building IMU measurements
for i=1:length(time)

   x_acc_meas(i) = S_accel(1,:)*acc_body(:,i) + x_acc_bias_true(i) + x_acc_noise*randn;

   y_acc_meas(i) = S_accel(2,:)*acc_body(:,i) + y_acc_bias_true(i) + y_acc_noise*randn;

   omega_r_meas(i) = S_gyro*omega_body(i) + omega_r_bias_true(i) + omega_r_noise*randn;

end

% Rounding the IMU data for true resolution
x_acc_meas = round(x_acc_meas*1000)/1000;
y_acc_meas = round(y_acc_meas*1000)/1000;

omega_r_meas = round(omega_r_meas*100)/100;

% Building the GPS measurements
for i=1:length(time_disc)

   j = (dt_disc/dt_cont)*i - (dt_disc/dt_cont - 1);

   DCM_t(1,1) = cos(y(j));
   DCM_t(1,2) = -sin(y(j));
   DCM_t(2,1) = sin(y(j));
   DCM_t(2,2) = cos(y(j));

   N_offset(i) = DCM_t(1,:)*offset;

   E_offset(i) = DCM_t(2,:)*offset;

   GPS_N_meas(i,1) = N_pos(j) + N_offset(i) + GPS_N_noise*randn;

   GPS_E_meas(i,1) = E_pos(j) + E_offset(i) + GPS_E_noise*randn;


end

% Rounding the GPS data
GPS_N_meas = round(GPS_N_meas*100)/100;
GPS_E_meas = round(GPS_E_meas*100)/100;

% Building the y_tilde vector
for i=1:length(time_disc)

   j = 2*i-1;
```

```
    k = 2*i;

    y_tilde(j,1) = GPS_N_meas(i,1);

    y_tilde(k,1) = GPS_E_meas(i,1);

end

GPS_data = [GPS_N_meas GPS_E_meas];

IMU_data = [x_acc_meas' y_acc_meas' omega_r_meas'];

save Initial_2D.mat

run EKF_2D
```

MatLab code that runs Runge-Kutta 4th order to produce the true motion for the 2-D case:

```
% Author:  Craig Odom

% Simulated truth code for 2-D case

function f = true_motion_2D(t,x)

global x_acc_eta y_acc_eta omega_r_eta amp w_length

% Initializing the functions
f = zeros(7,1);

% Equations of motion
f(1) = x(3);
f(2) = x(4);
f(3) = -4*amp*sin(2*pi*t/w_length)*pi^2/w_length^2;
f(4) = 0;
f(5) = x_acc_eta*randn;
f(6) = y_acc_eta*randn;
f(7) = omega_r_eta*randn;
```

MatLab code that runs the Kalman Filter algorithm for the 2-D case:

```
% Author:  Craig Odom

% Kalman Filter code that uses simulated measurements for 2-D case

clc;
clear;
```

```
close all;

load Initial_2D.mat

global x_acc_m y_acc_m omega_r_m g Q offset

% Initial Estimation Error Covariance Matrix
for i=1:11

   if i<=4

      P0(i,i) = 0.5;

   elseif i==5

      P0(i,i) = 0.005;

   elseif i==6||i==7

      P0(i,i) = 0.005;

   elseif i==8

      P0(i,i) = 0.005;

   elseif i==9||i==10

      P0(i,i) = 0.005;

   else

      P0(i,i) = 0.005;

   end

end

% Process Noise Covariance Matrix
x_acc_noise_mss = x_acc_noise*g;
y_acc_noise_mss = y_acc_noise*g;

omega_r_noise_rs = omega_r_noise*pi/180;

Q = [x_acc_noise_mss^2 0 0
   0 y_acc_noise_mss^2 0
   0 0 omega_r_noise_rs^2];

% Measurement Noise Covariance Matrix
```

```
R = [GPS_N_noise^2 0
   0 GPS_E_noise^2];

% Initial State Estimation
x_start = [0 0 25 5 0 0 0 0 ones(1,3)];

x_k_neg = x_start';

P_k_neg = P0;

step = dt_disc/dt_cont;

z = 1;

iteration = 0;

print = 0;

% Extended Kalman Filter
for i=1:length(time)

   % step builds until a new GPS measurement is available
   if step == dt_disc/dt_cont;

      j = 2*z-1;
      k = 2*z;

      % Direction Cosine Matrix
      DCM_ap(1,1) = cos(x_k_neg(5,1));
      DCM_ap(1,2) = sin(x_k_neg(5,1));
      DCM_ap(2,1) = -sin(x_k_neg(5,1));
      DCM_ap(2,2) = cos(x_k_neg(5,1));

      DCM_t_ap = DCM_ap';

      h(1,1) = x_k_neg(1,1) + DCM_t_ap(1,:)*offset;
      h(2,1) = x_k_neg(2,1) + DCM_t_ap(2,:)*offset;

      % Measurement Sensitivity Matrix
      H15 = -sin(x_k_neg(5,1))*x_offset-cos(x_k_neg(5,1))*y_offset;
      H25 = cos(x_k_neg(5,1))*x_offset-sin(x_k_neg(5,1))*y_offset;

      H = [1 0 0 0 H15 0 0 0 0 0 0
          0 1 0 0 H25 0 0 0 0 0 0];

      % Kalman Gain Matrix
      K = P_k_neg*H'*inv(H*P_k_neg*H' + R);
```

```
    Inn(:,i) = y_tilde(j:k,1) - h;

    % Correction to State Vector
    x_k_pos = x_k_neg + K*(Inn(:,i));

    % Correction to P Matrix
    P_k_pos = (eye(length(P0)) - K*H)*P_k_neg;

    step = 0;

    z = z + 1;

end

% Saving all state information
x_plot(:,i) = x_k_pos;

% Augmenting the IMU measurements from g's to m/s/s and deg/s to rad/s
x_acc_m = x_acc_meas(i)*g;

y_acc_m = y_acc_meas(i)*g;

omega_r_m = omega_r_meas(i)*pi/180;

% Setting the first part of the initial condition vector for Runge
% Kutta
x0(1:length(P0),1) = x_k_pos;

% Converting elements of P matrix to an array
for d=1:length(P0)

    j = length(P0)*d - (length(P0)-1);
    k = length(P0)*d;

    P_k_pos_col(j:k,1) = P_k_pos(d,1:length(P0));

end

% Setting the last part of the initial condition vector for Runge Kutta
x0(length(P0)+1:length(P0)+length(P0)^2,1) = P_k_pos_col;

% Calling the ode4 (Runge Kutta 4th order) integration tool
x_est = ode4(@Prop_2D,[0 0.01],x0);

% Taking the last part of the RK solution
P_est = x_est(length(x_est(:,1)),length(P0)+1:length(P0)+length(P0)^2);

% Converting the array P_est into the elements of P_k_neg
```

```
    for d=1:length(P0)

        j = length(P0)*d - (length(P0)-1);
        k = length(P0)*d;
        P_k_neg(d,1:length(P0)) = P_est(j:k);

    end

    % Converting the first part of the solution to x_k_neg
    x_k_neg = x_est(length(x_est(:,1)),1:length(P0))';

    % x_k_pos/P_k_pos are the best estimates available
    x_k_pos = x_k_neg;

    P_k_pos = P_k_neg;

    step = step + 1;

    % Printing out the simulation time every second
    if iteration == 100

        print = print + 1;

        Sim_time = print

        iteration = 0;

    end

    iteration = iteration + 1;

end

% Solving for the percentage error of the SFT estimate
for i=1:length(time)

    x_acc_per_error(i) = (S_a_x - 1/x_plot(9,i))/S_a_x*100;
    y_acc_per_error(i) = (S_a_y - 1/x_plot(10,i))/S_a_y*100;

    omega_r_per_error(i) = (S_g_r - 1/x_plot(11,i))/S_g_r*100;

end

% Saving all the data
save EKF_2D.mat

% Printing all the figures
figure (1)
```

```
plot(time,x_plot(1,:) - N_pos')
xlabel('Simulation Time [sec]')
ylabel('North Position Estimation Error [m]')
axis([0 200 -2 2])

figure (2)
plot(time,x_plot(2,:) - E_pos')
xlabel('Simulation Time [sec]')
ylabel('East Position Estimation Error [m]')
axis([0 200 -2 2])

figure (3)
plot(time,x_plot(3,:) - N_vel')
xlabel('Simulation Time [sec]')
ylabel('North Velocity Estimation Error [m/s]')

figure (4)
plot(time,x_plot(4,:) - E_vel')
xlabel('Simulation Time [sec]')
ylabel('East Velocity Estimation Error [m/s]')

figure (5)
plot(time,(yaw - x_plot(5,:))*180/pi)
xlabel('Simulation Time [sec]')
ylabel('Yaw Estimation Error [deg]')

figure (6)
plot(time,(x_acc_bias_true - x_plot(6,:)'/9.807)*1000)
xlabel('Simulation Time [sec]')
ylabel('x-axis Accelerometer Bias Estimation Error [mg]')

figure (7)
plot(time,(y_acc_bias_true - x_plot(7,:)'/9.807)*1000)
xlabel('Simulation Time [sec]')
ylabel('y-axis Accelerometer Bias Estimation Error [mg]')

figure (8)
plot(time,(omega_r_bias_true - x_plot(8,:)'*180/pi))
xlabel('Simulation Time [sec]')
ylabel('z-axis ARS Bias Estimation Error [deg/s]')

figure (9)
plot(time,x_acc_per_error)
xlabel('Simulation Time [sec]')
ylabel('x-axis Accelerometer SFT Error [%]')

figure (10)
plot(time,y_acc_per_error)
```

```
xlabel('Simulation Time [sec]')
ylabel('y-axis Accelerometer SFT Error [%]')

figure (11)
plot(time,omega_r_per_error)
xlabel('Simulation Time [sec]')
ylabel('z-axis ARS SFT Error [%]')

figure (12)
plot(time,x_acc_bias_true*1000)
xlabel('Simulation Time [sec]')
ylabel('True x-axis Accelerometer Bias [mg]')

figure (13)
plot(time,y_acc_bias_true*1000)
xlabel('Simulation Time [sec]')
ylabel('True y-axis Accelerometer Bias [mg]')

figure (14)
plot(time,omega_r_bias_true)
xlabel('Simulation Time [sec]')
ylabel('True z-axis ARS Bias [deg/s]')

figure (15)
plot(E_pos,N_pos)
xlabel('East Position of AGV')
ylabel('North Position of AGV')
axis([0 1000 -200 200])
```

MatLab code that runs Runge-Kutta 4th order to produce the *apriori* estimates for the 2-D case (including the estimation error covariance matrix):

```
% Author:  Craig Odom

% Propagation code for apriori solution for 2-D case

function f = Prop_2D(t,x)

global x_acc_m y_acc_m omega_r_m g Q

% Initialzing the functions
f = zeros(132,1);

cx5 = cos(x(5));
sx5 = sin(x(5));

% Truth model
```

```
w_r = x(11)*(omega_r_m - x(8));

f3x = cx5;
f3y = -sx5;
f4x = sx5;
f4y = cx5;

a_x = x(9)*(x_acc_m - x(6));
a_y = x(10)*(y_acc_m - x(7));

% Equations of motion
f(1) = x(3);
f(2) = x(4);
f(3) = f3x*a_x + f3y*a_y;
f(4) = f4x*a_x + f4y*a_y;
f(5) = w_r;
f(6) = 0;
f(7) = 0;
f(8) = 0;
f(9) = 0;
f(10) = 0;
f(11) = 0;

% Process Noise Mapping Matrix
G = [0 0 0
   0 0 0
   f3x*x(9) f3y*x(10) 0
   f4x*x(9) f4y*x(10) 0
   0 0 x(11)
   0 0 0
   0 0 0
   0 0 0
   0 0 0
   0 0 0
   0 0 0];

% Partial derivatives matrix
F35 = -sx5*x(9)*(x_acc_m-x(6))-cx5*x(10)*(y_acc_m-x(7));
F36 = -cx5*x(9);
F37 = sx5*x(10);
F38 = 0;
F39 = cx5*(x_acc_m-x(6));
F310 = -sx5*(y_acc_m-x(7));
F311 = 0;
F45 = cx5*x(9)*(x_acc_m-x(6))-sx5*x(10)*(y_acc_m-x(7));
F46 = -sx5*x(9);
F47 = -cx5*x(10);
F48 = 0;
```

```
F49 = sx5*(x_acc_m-x(6));
F410 = cx5*(y_acc_m-x(7));
F411 = 0;
F55 = 0;
F56 = 0;
F57 = 0;
F58 = -x(11);
F59 = 0;
F510 = 0;
F511 = omega_r_m-x(8);

F = [0 0 1 0 0 0 0 0 0 0 0
    0 0 0 1 0 0 0 0 0 0 0
    0 0 0 0 F35 F36 F37 F38 F39 F310 F311
    0 0 0 0 F45 F46 F47 F48 F49 F410 F411
    0 0 0 0 F55 F56 F57 F58 F59 F510 F511
    0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0];

st = length(F) + 1;

% Filling the P matrix with the states
for i=1:length(F)

   for j=1:length(F)

      P(i,j) = x(st);

      st = st + 1;

   end

end

% Variation of Ricatti equation
P_dot = F*P + P*F' + G*Q*G';

st = length(F) + 1;

% Filling the functions with the elements of P_dot
for i=1:length(F)

   for j=1:length(F)
```

```
        f(st) = P_dot(i,j);

        st = st + 1;

    end

end
```

MatLAB code for simulating 3-D IMU/GPS measurements:

```
% Author:  Craig Odom

% Simulated measurement code for 3-D case

clc;
clear;
close all;

global x_acc_eta y_acc_eta z_acc_eta omega_p_eta omega_q_eta omega_r_eta g Q offset amp
w_length

g = 9.807;

% Noise parameters

GPS_N_noise = 1;
GPS_E_noise = 1;
GPS_D_noise = 1;
x_acc_noise = 0.005;
y_acc_noise = 0.005;
z_acc_noise = 0.005;
omega_p_noise = 0.05;
omega_q_noise = 0.05;
omega_r_noise = 0.05;
x_acc_eta = 0.0001;
y_acc_eta = 0.0001;
z_acc_eta = 0.0001;
omega_p_eta = 0.002;
omega_q_eta = 0.002;
omega_r_eta = 0.002;

% IMU--->GPS offsets

x_offset = -0.67;
y_offset = 0;
z_offset = -0.9;
```

```
offset = [x_offset;y_offset;z_offset];

% Scale factor actual and nominal (V/g, V/(deg/s))

S_act_x_acc = 2.57;
S_act_y_acc = 2.56;
S_act_z_acc = 2.44;
S_act_omega_p = 0.103;
S_act_omega_q = 0.1028;
S_act_omega_r = 0.097;

S_nom_x_acc = 0.4;
S_nom_y_acc = 0.4;
S_nom_z_acc = 0.4;
S_nom_omega_p = 10;
S_nom_omega_q = 10;
S_nom_omega_r = 10;

% Time parameters

t_begin = 0;
dt_cont = 0.01;
dt_disc = 0.05;
t_final = 300;
time = t_begin:dt_cont:t_final;
time_disc = t_begin:dt_disc:t_final;

% True motion

amp = 200;
w_length = 50;

N_pos = 0;
E_pos = 0;
D_pos = 0;
N_vel = 2*amp*pi/w_length;
E_vel = 5;
D_vel = 0;
yaw = 0;
pitch = 0;
roll = 0;
x_acc_bias_true = 0.02;
y_acc_bias_true = 0.02;
z_acc_bias_true = 0.02;
omega_p_bias_true = 0.1;
omega_q_bias_true = 0.1;
omega_r_bias_true = 0.1;
```

```
x0_true_motion(1:8,1) = [N_pos E_pos D_pos N_vel E_vel D_vel pitch roll]';
x0_true_motion(9:14,1) = [x_acc_bias_true y_acc_bias_true z_acc_bias_true
omega_p_bias_true omega_q_bias_true omega_r_bias_true]';

N_acc = -4*amp*sin(2*pi*time/w_length)*pi^2/w_length^2;

E_acc = zeros(1,length(time));

D_acc = 0.5*cos(time/3) - 0.2*cos(time/5);

% Calling the ode4 function (Runge Kutta 4th order)
x_true = ode4(@true_motion_3D,time,x0_true_motion);

N_pos = x_true(:,1);
E_pos = x_true(:,2);
D_pos = x_true(:,3);
N_vel = x_true(:,4);
E_vel = x_true(:,5);
D_vel = x_true(:,6);
pitch = x_true(:,7);
roll = x_true(:,8);
x_acc_bias_true = x_true(:,9);
y_acc_bias_true = x_true(:,10);
z_acc_bias_true = x_true(:,11);
omega_p_bias_true = x_true(:,12);
omega_q_bias_true = x_true(:,13);
omega_r_bias_true = x_true(:,14);

pitch_rate = 0.01*sin(time/2) - 0.02*cos(time/3);

roll_rate = 0.02*sin(time) - 0.03*cos(time/2);

% Building the true yaw motion
for i=1:length(time)-1

   yaw(i+1) = atan2(E_vel(i+1),N_vel(i+1));

   yaw_rate(i) = (E_acc(i)*N_vel(i) - N_acc(i)*E_vel(i))/(E_vel(i)^2 + N_vel(i)^2);

   if i==length(time)-1

      yaw_rate(i+1) = (E_acc(i+1)*N_vel(i+1) - N_acc(i+1)*E_vel(i+1))/(E_vel(i+1)^2 +
N_vel(i+1)^2);

   end

end
```

```
y = yaw;
p = pitch;
r = roll;

% Building the true rotations
for i=1:length(time)

    omega_p(i) = -sin(p(i))*yaw_rate(i) + roll_rate(i);

    omega_q(i) = cos(p(i))*sin(r(i))*yaw_rate(i) + cos(r(i))*pitch_rate(i);

    omega_r(i) = cos(p(i))*cos(r(i))*yaw_rate(i) - sin(r(i))*pitch_rate(i);

end

% Direction Cosine Matrix
for i=1:length(time)

    DCM(1,1) = cos(p(i))*cos(y(i));
    DCM(1,2) = cos(p(i))*sin(y(i));
    DCM(1,3) = -sin(p(i));
    DCM(2,1) = sin(p(i))*sin(r(i))*cos(y(i)) - cos(r(i))*sin(y(i));
    DCM(2,2) = sin(p(i))*sin(r(i))*sin(y(i)) + cos(r(i))*cos(y(i));
    DCM(2,3) = cos(p(i))*sin(r(i));
    DCM(3,1) = sin(p(i))*cos(r(i))*cos(y(i)) + sin(r(i))*sin(y(i));
    DCM(3,2) = sin(p(i))*cos(r(i))*sin(y(i)) - sin(r(i))*cos(y(i));
    DCM(3,3) = cos(p(i))*cos(r(i));

    x_acc_body(i) = DCM(1,:)*[N_acc(i) E_acc(i) D_acc(i)]';

    y_acc_body(i) = DCM(2,:)*[N_acc(i) E_acc(i) D_acc(i)]';

    z_acc_body(i) = DCM(3,:)*[N_acc(i) E_acc(i) D_acc(i)]';

end

x_acc_body = x_acc_body/g;
y_acc_body = y_acc_body/g;
z_acc_body = z_acc_body/g;

omega_p = omega_p*180/pi;
omega_q = omega_q*180/pi;
omega_r = omega_r*180/pi;

acc_body = [x_acc_body;y_acc_body;z_acc_body];

omega_body = [omega_p;omega_q;omega_r];
```

```
% Scale factor errors
S_a_x = S_act_x_acc*S_nom_x_acc;
S_a_y = S_act_y_acc*S_nom_y_acc;
S_a_z = S_act_z_acc*S_nom_z_acc;

S_g_p = S_act_omega_p*S_nom_omega_p;
S_g_q = S_act_omega_q*S_nom_omega_q;
S_g_r = S_act_omega_r*S_nom_omega_r;

S_accel = [S_a_x S_a_y S_a_z];

S_gyro = [S_g_p S_g_q S_g_r];

% Building IMU measurements
for i=1:length(time)

   x_acc_meas(i) = S_accel(1)*(acc_body(1,i) + sin(p(i))) + x_acc_bias_true(i) +
x_acc_noise*randn;

   y_acc_meas(i) = S_accel(2)*(acc_body(2,i) - cos(p(i))*sin(r(i))) + y_acc_bias_true(i) +
y_acc_noise*randn;

   z_acc_meas(i) = S_accel(3)*(acc_body(3,i) - cos(p(i))*cos(r(i))) + z_acc_bias_true(i) +
z_acc_noise*randn;

   omega_p_meas(i) = S_gyro(1)*omega_body(1,i) + omega_p_bias_true(i) +
omega_p_noise*randn;

   omega_q_meas(i) = S_gyro(2)*omega_body(2,i) + omega_q_bias_true(i) +
omega_q_noise*randn;

   omega_r_meas(i) = S_gyro(3)*omega_body(3,i) + omega_r_bias_true(i) +
omega_r_noise*randn;

end

% Rounding the IMU data for true resolution
x_acc_meas = round(x_acc_meas*1000)/1000;
y_acc_meas = round(y_acc_meas*1000)/1000;
z_acc_meas = round(z_acc_meas*1000)/1000;

omega_p_meas = round(omega_p_meas*100)/100;
omega_q_meas = round(omega_q_meas*100)/100;
omega_r_meas = round(omega_r_meas*100)/100;

% Building the GPS measurements
for i=1:length(time_disc)
```

```
    j = (dt_disc/dt_cont)*i - (dt_disc/dt_cont - 1);

    DCM_t(1,1) = cos(p(j))*cos(y(j));
    DCM_t(1,2) = sin(p(j))*sin(r(j))*cos(y(j)) - cos(r(j))*sin(y(j));
    DCM_t(1,3) = sin(p(j))*cos(r(j))*cos(y(j)) + sin(r(j))*sin(y(j));
    DCM_t(2,1) = cos(p(j))*sin(y(j));
    DCM_t(2,2) = sin(p(j))*sin(r(j))*sin(y(j)) + cos(r(j))*cos(y(j));
    DCM_t(2,3) = sin(p(j))*cos(r(j))*sin(y(j)) - sin(r(j))*cos(y(j));
    DCM_t(3,1) = -sin(p(j));
    DCM_t(3,2) = cos(p(j))*sin(r(j));
    DCM_t(3,3) = cos(p(j))*cos(r(j));

    N_offset(i) = DCM_t(1,1:3)*offset;

    E_offset(i) = DCM_t(2,1:3)*offset;

    D_offset(i) = DCM_t(3,1:3)*offset;

    GPS_N_meas(i,1) = N_pos(j) + N_offset(i) + GPS_N_noise*randn;

    GPS_E_meas(i,1) = E_pos(j) + E_offset(i) + GPS_E_noise*randn;

    GPS_D_meas(i,1) = D_pos(j) + D_offset(i) + GPS_D_noise*randn;

end

% Rounding the GPS data
GPS_N_meas = round(GPS_N_meas*100)/100;
GPS_E_meas = round(GPS_E_meas*100)/100;
GPS_D_meas = round(GPS_D_meas*100)/100;

% Building the y_tilde vector
for i=1:length(time_disc)

    j = 3*i-2;
    k = 3*i-1;
    w = 3*i;

    y_tilde(j,1) = GPS_N_meas(i,1);

    y_tilde(k,1) = GPS_E_meas(i,1);

    y_tilde(w,1) = GPS_D_meas(i,1);

end

GPS_data = [GPS_N_meas GPS_E_meas GPS_D_meas];
```

```
IMU_data = [x_acc_meas' y_acc_meas' z_acc_meas' omega_p_meas' omega_q_meas'
omega_r_meas'];

save Initial_3D.mat

run EKF_3D
```

MatLab code that runs Runge-Kutta 4th order to produce the true motion for the 3-D case:

```
% Author:  Craig Odom

% Simulated truth code for 3-D case

function f = true_motion_3D(t,x)

global x_acc_eta y_acc_eta z_acc_eta omega_p_eta omega_q_eta omega_r_eta amp w_length

% Initialzing the functions
f = zeros(14,1);

% Equations of motion
f(1) = x(4);
f(2) = x(5);
f(3) = x(6);
f(4) = -4*amp*sin(2*pi*t/w_length)*pi^2/w_length^2;
f(5) = 0;
f(6) = 0.5*cos(t/3) - 0.2*cos(t/5);
f(7) = 0.01*sin(t/2) - 0.02*cos(t/3);
f(8) = 0.02*sin(t) - 0.03*cos(t/2);
f(9) = x_acc_eta*randn;
f(10) = y_acc_eta*randn;
f(11) = z_acc_eta*randn;
f(12) = omega_p_eta*randn;
f(13) = omega_q_eta*randn;
f(14) = omega_r_eta*randn;
```

MatLab code that runs the Kalman Filter algorithm for the 3-D case:

```
% Author:  Craig Odom

% Kalman Filter code that uses simulated measurements for 3-D case

clc;
clear;
close all;
```

```
load Initial_3D.mat

global x_acc_m y_acc_m z_acc_m omega_p_m omega_q_m omega_r_m g Q offset

% Initial Estimation Error Covariance Matrix
for i=1:21

   if i<=6

      P0(i,i) = 0.5;

   elseif i==7||i==8||i==9

      P0(i,i) = 0.05;

   elseif i==10||i==11||i==12

      P0(i,i) = 0.05;

   elseif i==13||i==14||i==15

      P0(i,i) = 0.005;

   elseif i==16||i==17||i==18

      P0(i,i) = 0.05;

   else

      P0(i,i) = 0.005;

   end

end

% Process Noise Covariance Matrix
x_acc_noise_mss = x_acc_noise*g;
y_acc_noise_mss = y_acc_noise*g;
z_acc_noise_mss = z_acc_noise*g;

omega_p_noise_rs = omega_p_noise*pi/180;
omega_q_noise_rs = omega_q_noise*pi/180;
omega_r_noise_rs = omega_r_noise*pi/180;

Q = [x_acc_noise_mss^2 0 0 0 0 0
   0 y_acc_noise_mss^2 0 0 0 0
   0 0 z_acc_noise_mss^2 0 0 0
   0 0 0 omega_p_noise_rs^2 0 0
```

```
   0 0 0 0 omega_q_noise_rs^2 0
   0 0 0 0 0 omega_r_noise_rs^2];

% Measurement Noise Covariance Matrix
R = [GPS_N_noise^2 0 0
   0 GPS_E_noise^2 0
   0 0 GPS_D_noise^2];

% Initial State Estimation
x_start = [0 0 0 25 5 0 0 0 0 0 0 0 0 0 0 ones(1,6)];

x_k_neg = x_start';

P_k_neg = P0;

step = dt_disc/dt_cont;

z = 1;

iteration = 0;

print = 0;

% Extended Kalman Filter
for i=1:length(time)

   % step builds until a new GPS measurement is available
   if step == dt_disc/dt_cont;

      j = 3*z-2;
      k = 3*z-1;
      w = 3*z;

      % Direction Cosine Matrix
      DCM_ap(1,1) = cos(x_k_neg(8,1))*cos(x_k_neg(7,1));
      DCM_ap(1,2) = cos(x_k_neg(8,1))*sin(x_k_neg(7,1));
      DCM_ap(1,3) = -sin(x_k_neg(8,1));
      DCM_ap(2,1) = sin(x_k_neg(8,1))*sin(x_k_neg(9,1))*cos(x_k_neg(7,1)) -
cos(x_k_neg(9,1))*sin(x_k_neg(7,1));
      DCM_ap(2,2) = sin(x_k_neg(8,1))*sin(x_k_neg(9,1))*sin(x_k_neg(7,1)) +
cos(x_k_neg(9,1))*cos(x_k_neg(7,1));
      DCM_ap(2,3) = cos(x_k_neg(8,1))*sin(x_k_neg(9,1));
      DCM_ap(3,1) = sin(x_k_neg(8,1))*cos(x_k_neg(9,1))*cos(x_k_neg(7,1)) +
sin(x_k_neg(9,1))*sin(x_k_neg(7,1));
      DCM_ap(3,2) = sin(x_k_neg(8,1))*cos(x_k_neg(9,1))*sin(x_k_neg(7,1)) -
sin(x_k_neg(9,1))*cos(x_k_neg(7,1));
      DCM_ap(3,3) = cos(x_k_neg(8,1))*cos(x_k_neg(9,1));
```

```
        DCM_t_ap = DCM_ap';

        h(1,1) = x_k_neg(1,1) + DCM_t_ap(1,1:3)*offset;
        h(2,1) = x_k_neg(2,1) + DCM_t_ap(2,1:3)*offset;
        h(3,1) = x_k_neg(3,1) + DCM_t_ap(3,1:3)*offset;

        % Measurement Sensitivity Matrix
        H17 = -cos(x_k_neg(8,1))*sin(x_k_neg(7,1))*x_offset+(-
sin(x_k_neg(8,1))*sin(x_k_neg(9,1))*sin(x_k_neg(7,1))-
cos(x_k_neg(9,1))*cos(x_k_neg(7,1)))*y_offset+(-
sin(x_k_neg(8,1))*cos(x_k_neg(9,1))*sin(x_k_neg(7,1))+sin(x_k_neg(9,1))*cos(x_k_neg(7,1)))
*z_offset;
        H18 = -
sin(x_k_neg(8,1))*cos(x_k_neg(7,1))*x_offset+cos(x_k_neg(8,1))*sin(x_k_neg(9,1))*cos(x_k_
neg(7,1))*y_offset+cos(x_k_neg(8,1))*cos(x_k_neg(9,1))*cos(x_k_neg(7,1))*z_offset;
        H19 =
(sin(x_k_neg(8,1))*cos(x_k_neg(9,1))*cos(x_k_neg(7,1))+sin(x_k_neg(9,1))*sin(x_k_neg(7,1))
)*y_offset+(-
sin(x_k_neg(8,1))*sin(x_k_neg(9,1))*cos(x_k_neg(7,1))+cos(x_k_neg(9,1))*sin(x_k_neg(7,1)))
*z_offset;
        H27 =
cos(x_k_neg(8,1))*cos(x_k_neg(7,1))*x_offset+(sin(x_k_neg(8,1))*sin(x_k_neg(9,1))*cos(x_k_
neg(7,1))-
cos(x_k_neg(9,1))*sin(x_k_neg(7,1)))*y_offset+(sin(x_k_neg(8,1))*cos(x_k_neg(9,1))*cos(x_k
_neg(7,1))+sin(x_k_neg(9,1))*sin(x_k_neg(7,1)))*z_offset;
        H28 = -
sin(x_k_neg(8,1))*sin(x_k_neg(7,1))*x_offset+cos(x_k_neg(8,1))*sin(x_k_neg(9,1))*sin(x_k_n
eg(7,1))*y_offset+cos(x_k_neg(8,1))*cos(x_k_neg(9,1))*sin(x_k_neg(7,1))*z_offset;
        H29 = (sin(x_k_neg(8,1))*cos(x_k_neg(9,1))*sin(x_k_neg(7,1))-
sin(x_k_neg(9,1))*cos(x_k_neg(7,1)))*y_offset+(-
sin(x_k_neg(8,1))*sin(x_k_neg(9,1))*sin(x_k_neg(7,1))-
cos(x_k_neg(9,1))*cos(x_k_neg(7,1)))*z_offset;
        H37 = 0;
        H38 = -cos(x_k_neg(8,1))*x_offset-sin(x_k_neg(8,1))*sin(x_k_neg(9,1))*y_offset-
sin(x_k_neg(8,1))*cos(x_k_neg(9,1))*z_offset;
        H39 = cos(x_k_neg(8,1))*cos(x_k_neg(9,1))*y_offset-
cos(x_k_neg(8,1))*sin(x_k_neg(9,1))*z_offset;

        H = [1 0 0 0 0 0 H17 H18 H19 0 0 0 0 0 0 0 0 0 0 0
             0 1 0 0 0 0 H27 H28 H29 0 0 0 0 0 0 0 0 0 0 0
             0 0 1 0 0 0 H37 H38 H39 0 0 0 0 0 0 0 0 0 0 0];

        % Kalman Gain Matrix
        K = P_k_neg*H'*inv(H*P_k_neg*H' + R);

        Inn(:,i) = y_tilde(j:w,1) - h;

        % Correction to State Vector
```

```
    x_k_pos = x_k_neg + K*(Inn(:,i));

    % Correction to P Matrix
    P_k_pos = (eye(length(P0)) - K*H)*P_k_neg;

    step = 0;

    z = z + 1;

end

% Saving all State information
x_plot(:,i) = x_k_pos;

% Augmenting the IMU measurements from g's to m/s/s and deg/s to rad/s
x_acc_m = x_acc_meas(i)*g;

y_acc_m = y_acc_meas(i)*g;

z_acc_m = z_acc_meas(i)*g;

omega_p_m = omega_p_meas(i)*pi/180;

omega_q_m = omega_q_meas(i)*pi/180;

omega_r_m = omega_r_meas(i)*pi/180;

% Setting the first part of the initial condition vector for Runge
% Kutta
x0(1:length(P0),1) = x_k_pos;

% Converting elements of P matrix to an array
for d=1:length(P0)

    j = length(P0)*d - (length(P0)-1);
    k = length(P0)*d;

    P_k_pos_col(j:k,1) = P_k_pos(d,1:length(P0));

end

% Setting the last part of the initial condition vector for Runge Kutta
x0(length(P0)+1:length(P0)+length(P0)^2,1) = P_k_pos_col;

% Calling the ode4 (Runge Kutta 4th order) integration tool
x_est = ode4(@Prop_3D,[0 0.01],x0);

% Taking the last part of the RK solution
```

```
    P_est = x_est(length(x_est(:,1)),length(P0)+1:length(P0)+length(P0)^2);

    % Converting the array P_est into the elements of P_k_neg
    for d=1:length(P0)

        j = length(P0)*d - (length(P0)-1);
        k = length(P0)*d;
        P_k_neg(d,1:length(P0)) = P_est(j:k);

    end

    % Converting the first part of the solution to x_k_neg
    x_k_neg = x_est(length(x_est(:,1)),1:length(P0))';

    % x_k_pos/P_k_pos are the best estimates available
    x_k_pos = x_k_neg;

    P_k_pos = P_k_neg;

    step = step + 1;

    % Printing out the simulation time every second
    if iteration == 100

        print = print + 1;

        Sim_time = print

        iteration = 0;

    end

    iteration = iteration + 1;

end

% Solving for the percentage error of the SFT estimate
for i=1:length(time)

    x_acc_per_error(i) = (S_a_x - 1/x_plot(16,i))/S_a_x*100;
    y_acc_per_error(i) = (S_a_y - 1/x_plot(17,i))/S_a_y*100;
    z_acc_per_error(i) = (S_a_z - 1/x_plot(18,i))/S_a_z*100;

    omega_p_per_error(i) = (S_g_p - 1/x_plot(19,i))/S_g_p*100;
    omega_q_per_error(i) = (S_g_q - 1/x_plot(20,i))/S_g_q*100;
    omega_r_per_error(i) = (S_g_r - 1/x_plot(21,i))/S_g_r*100;

end
```

```
% Saving all the data
save EKF_3D.mat

% Printing all the figures
figure (1)
plot(time,x_plot(1,:) - N_pos')
xlabel('Simulation Time [sec]')
ylabel('North Position Estimation Error [m]')
axis([0 300 -2 2])

figure (2)
plot(time,x_plot(2,:) - E_pos')
xlabel('Simulation Time [sec]')
ylabel('East Position Estimation Error [m]')
axis([0 300 -2 2])

figure (3)
plot(time,x_plot(3,:) - D_pos')
xlabel('Simulation Time [sec]')
ylabel('Down Position Estimation Error [m]')
axis([0 300 -2 2])

figure (4)
plot(time,x_plot(4,:) - N_vel')
xlabel('Simulation Time [sec]')
ylabel('North Velocity Estimation Error [m/s]')

figure (5)
plot(time,x_plot(5,:) - E_vel')
xlabel('Simulation Time [sec]')
ylabel('East Velocity Estimation Error [m/s]')

figure (6)
plot(time,x_plot(6,:) - D_vel')
xlabel('Simulation Time [sec]')
ylabel('Down Velocity Estimation Error [m/s]')

figure (7)
plot(time,(yaw - x_plot(7,:))*180/pi)
xlabel('Simulation Time [sec]')
ylabel('Yaw Estimation Error [deg]')

figure (8)
plot(time,(pitch' - x_plot(8,:))*180/pi)
xlabel('Simulation Time [sec]')
ylabel('Pitch Estimation Error [deg]')
```

```
figure (9)
plot(time,(roll' - x_plot(9,:))*180/pi)
xlabel('Simulation Time [sec]')
ylabel('Roll Estimation Error [deg]')

figure (10)
plot(time,(x_acc_bias_true - x_plot(10,:)'/9.807)*1000)
xlabel('Simulation Time [sec]')
ylabel('x-axis Accelerometer Bias Estimation Error [mg]')

figure (11)
plot(time,(y_acc_bias_true - x_plot(11,:)'/9.807)*1000)
xlabel('Simulation Time [sec]')
ylabel('y-axis Accelerometer Bias Estimation Error [mg]')

figure (12)
plot(time,(z_acc_bias_true - x_plot(12,:)'/9.807)*1000)
xlabel('Simulation Time [sec]')
ylabel('z-axis Accelerometer Bias Estimation Error [mg]')

figure (13)
plot(time,(omega_p_bias_true - x_plot(13,:)'*180/pi))
xlabel('Simulation Time [sec]')
ylabel('x-axis ARS Bias Estimation Error [deg/s]')

figure (14)
plot(time,(omega_q_bias_true - x_plot(14,:)'*180/pi))
xlabel('Simulation Time [sec]')
ylabel('y-axis ARS Bias Estimation Error [deg/s]')

figure (15)
plot(time,(omega_r_bias_true - x_plot(15,:)'*180/pi))
xlabel('Simulation Time [sec]')
ylabel('z-axis ARS Bias Estimation Error [deg/s]')

figure (16)
plot(time,x_acc_per_error)
xlabel('Simulation Time [sec]')
ylabel('x-axis Accelerometer SFT Error [%]')

figure (17)
plot(time,y_acc_per_error)
xlabel('Simulation Time [sec]')
ylabel('y-axis Accelerometer SFT Error [%]')

figure (18)
plot(time,z_acc_per_error)
xlabel('Simulation Time [sec]')
```

```
ylabel('z-axis Accelerometer SFT Error [%]')

figure (19)
plot(time,omega_p_per_error)
xlabel('Simulation Time [sec]')
ylabel('x-axis ARS SFT Error [%]')

figure (20)
plot(time,omega_q_per_error)
xlabel('Simulation Time [sec]')
ylabel('y-axis ARS SFT Error [%]')

figure (21)
plot(time,omega_r_per_error)
xlabel('Simulation Time [sec]')
ylabel('z-axis ARS SFT Error [%]')

figure (22)
plot(time,x_acc_bias_true*1000)
xlabel('Simulation Time [sec]')
ylabel('True x-axis Accelerometer Bias [mg]')

figure (23)
plot(time,y_acc_bias_true*1000)
xlabel('Simulation Time [sec]')
ylabel('True y-axis Accelerometer Bias [mg]')

figure (24)
plot(time,z_acc_bias_true*1000)
xlabel('Simulation Time [sec]')
ylabel('True z-axis Accelerometer Bias [mg]')

figure (25)
plot(time,omega_p_bias_true)
xlabel('Simulation Time [sec]')
ylabel('True x-axis ARS Bias [deg/s]')

figure (26)
plot(time,omega_q_bias_true)
xlabel('Simulation Time [sec]')
ylabel('True y-axis ARS Bias [deg/s]')

figure (27)
plot(time,omega_r_bias_true)
xlabel('Simulation Time [sec]')
ylabel('True z-axis ARS Bias [deg/s]')

figure (28)
```

```
plot(time,D_pos)
xlabel('Simulation Time [sec]')
ylabel('True Down Position of AGV [m]')

figure (29)
plot(time,pitch*180/pi)
xlabel('Simulation Time [sec]')
ylabel('True Pitch of AGV [deg]')

figure (30)
plot(time,roll*180/pi)
xlabel('Simulation Time [sec]')
ylabel('True Roll of AGV [deg]')
```

MatLab code that runs Runge-Kutta 4th order to produce the *apriori* estimates for the 3-D case (including the estimation error covariance matrix):

```
% Author:  Craig Odom

% Propagation code for apriori solution for 3-D case

function f = Prop_3D(t,x)

global x_acc_m y_acc_m z_acc_m omega_p_m omega_q_m omega_r_m g Q

% Initialzing the functions
f = zeros(462,1);

cx7 = cos(x(7));
sx7 = sin(x(7));
tx7 = tan(x(7));
cx8 = cos(x(8));
sx8 = sin(x(8));
tx8 = tan(x(8));
cx9 = cos(x(9));
sx9 = sin(x(9));
tx9 = tan(x(9));

% Truth model
w_p = x(19)*(omega_p_m - x(13));
w_q = x(20)*(omega_q_m - x(14));
w_r = x(21)*(omega_r_m - x(15));

f4x = cx7*cx8;
f4y = sx9*sx8*cx7-cx9*sx7;
f4z = cx9*sx8*cx7+sx9*sx7;
f5x = sx7*cx8;
```

```
f5y = sx9*sx8*sx7+cx9*cx7;
f5z = cx9*sx8*sx7-sx9*cx7;
f6x = -sx8;
f6y = sx9*cx8;
f6z = cx9*cx8;

a_x = x(16)*(x_acc_m - x(10)) - g*sx8;
a_y = x(17)*(y_acc_m - x(11)) + g*cx8*sx9;
a_z = x(18)*(z_acc_m - x(12)) + g*cx8*cx9;

% Equations of motion
f(1) = x(4);
f(2) = x(5);
f(3) = x(6);
f(4) = f4x*a_x + f4y*a_y + f4z*a_z;
f(5) = f5x*a_x + f5y*a_y + f5z*a_z;
f(6) = f6x*a_x + f6y*a_y + f6z*a_z;
f(7) = 1/cx8*(sx9*w_q+cx9*w_r);
f(8) = cx9*w_q - sx9*w_r;
f(9) = w_p + tx8*(sx9*w_q+cx9*w_r);
f(10) = 0;
f(11) = 0;
f(12) = 0;
f(13) = 0;
f(14) = 0;
f(15) = 0;
f(16) = 0;
f(17) = 0;
f(18) = 0;
f(19) = 0;
f(20) = 0;
f(21) = 0;

% Process Noise Mapping Matrix
G = [0 0 0 0 0 0
   0 0 0 0 0 0
   0 0 0 0 0 0
   f4x*x(16) f4y*x(17) f4z*x(18) 0 0 0
   f5x*x(16) f5y*x(17) f5z*x(18) 0 0 0
   f6x*x(16) f6y*x(17) f6z*x(18) 0 0 0
   0 0 0 0 1/cx8*sx9*x(20) 1/cx8*cx9*x(21)
   0 0 0 0 cx9*x(20) -sx9*x(21)
   0 0 0 x(19) tx8*sx9*x(20) tx8*cx9*x(21)
   0 0 0 0 0 0
   0 0 0 0 0 0
   0 0 0 0 0 0
   0 0 0 0 0 0
   0 0 0 0 0 0
```

```
    0 0 0 0 0 0
    0 0 0 0 0 0
    0 0 0 0 0 0
    0 0 0 0 0 0
    0 0 0 0 0 0
    0 0 0 0 0 0
    0 0 0 0 0 0];

% Partial derivatives matrix
F47 = -sx7*cx8*(x(16)*(x_acc_m-x(10))-g*sx8)+(-sx9*sx8*sx7-cx9*cx7)*(x(17)*(y_acc_m-
x(11))+g*cx8*sx9)+(-cx9*sx8*sx7+sx9*cx7)*(x(18)*(z_acc_m-x(12))+g*cx8*cx9);
F48 = -cx7*sx8*(x(16)*(x_acc_m-x(10))-g*sx8)-cx7*cx8^2*g+sx9*cx8*cx7*(x(17)*(y_acc_m-
x(11))+g*cx8*sx9)-(sx9*sx8*cx7-cx9*sx7)*g*sx8*sx9+cx9*cx8*cx7*(x(18)*(z_acc_m-
x(12))+g*cx8*cx9)-(cx9*sx8*cx7+sx9*sx7)*g*sx8*cx9;
F49 = (cx9*sx8*cx7+sx9*sx7)*(x(17)*(y_acc_m-x(11))+g*cx8*sx9)+(sx9*sx8*cx7-
cx9*sx7)*g*cx8*cx9+(-sx9*sx8*cx7+cx9*sx7)*(x(18)*(z_acc_m-x(12))+g*cx8*cx9)-
(cx9*sx8*cx7+sx9*sx7)*g*cx8*sx9;
F410 = -cx7*cx8*x(16);
F411 = -(sx9*sx8*cx7-cx9*sx7)*x(17);
F412 = -(cx9*sx8*cx7+sx9*sx7)*x(18);
F413 = 0;
F414 = 0;
F415 = 0;
F416 = cx7*cx8*(x_acc_m-x(10));
F417 = (sx9*sx8*cx7-cx9*sx7)*(y_acc_m-x(11));
F418 = (cx9*sx8*cx7+sx9*sx7)*(z_acc_m-x(12));
F419 = 0;
F420 = 0;
F421 = 0;
F57 = cx7*cx8*(x(16)*(x_acc_m-x(10))-g*sx8)+(sx9*sx8*cx7-cx9*sx7)*(x(17)*(y_acc_m-
x(11))+g*cx8*sx9)+(cx9*sx8*cx7+sx9*sx7)*(x(18)*(z_acc_m-x(12))+g*cx8*cx9);
F58 = -sx7*sx8*(x(16)*(x_acc_m-x(10))-g*sx8)-sx7*cx8^2*g+sx9*cx8*sx7*(x(17)*(y_acc_m-
x(11))+g*cx8*sx9)-(sx9*sx8*sx7+cx9*cx7)*g*sx8*sx9+cx9*cx8*sx7*(x(18)*(z_acc_m-
x(12))+g*cx8*cx9)-(cx9*sx8*sx7-sx9*cx7)*g*sx8*cx9;
F59 = (cx9*sx8*sx7-sx9*cx7)*(x(17)*(y_acc_m-
x(11))+g*cx8*sx9)+(sx9*sx8*sx7+cx9*cx7)*g*cx8*cx9+(-sx9*sx8*sx7-
cx9*cx7)*(x(18)*(z_acc_m-x(12))+g*cx8*cx9)-(cx9*sx8*sx7-sx9*cx7)*g*cx8*sx9;
F510 = -sx7*cx8*x(16);
F511 = -(sx9*sx8*sx7+cx9*cx7)*x(17);
F512 = -(cx9*sx8*sx7-sx9*cx7)*x(18);
F513 = 0;
F514 = 0;
F515 = 0;
F516 = sx7*cx8*(x_acc_m-x(10));
F517 = (sx9*sx8*sx7+cx9*cx7)*(y_acc_m-x(11));
F518 = (cx9*sx8*sx7-sx9*cx7)*(z_acc_m-x(12));
F519 = 0;
F520 = 0;
```

```
F521 = 0;
F67 = 0;
F68 = -cx8*(x(16)*(x_acc_m-x(10))-g*sx8)+sx8*g*cx8-sx9*sx8*(x(17)*(y_acc_m-
x(11))+g*cx8*sx9)-sx9^2*cx8*g*sx8-cx9*sx8*(x(18)*(z_acc_m-x(12))+g*cx8*cx9)-
cx9^2*cx8*g*sx8;
F69 = cx9*cx8*(x(17)*(y_acc_m-x(11))+g*cx8*sx9)-sx9*cx8*(x(18)*(z_acc_m-
x(12))+g*cx8*cx9);
F610 = sx8*x(16);
F611 = -sx9*cx8*x(17);
F612 = -cx9*cx8*x(18);
F613 = 0;
F614 = 0;
F615 = 0;
F616 = -sx8*(x_acc_m-x(10));
F617 = sx9*cx8*(y_acc_m-x(11));
F618 = cx9*cx8*(z_acc_m-x(12));
F619 = 0;
F620 = 0;
F621 = 0;
F77 = 0;
F78 = 1/cx8^2*(sx9*x(20)*(omega_q_m-x(14))+cx9*x(21)*(omega_r_m-x(15)))*sx8;
F79 = 1/cx8*(cx9*x(20)*(omega_q_m-x(14))-sx9*x(21)*(omega_r_m-x(15)));
F710 = 0;
F711 = 0;
F712 = 0;
F713 = 0;
F714 = -1/cx8*sx9*x(20);
F715 = -1/cx8*cx9*x(21);
F716 = 0;
F717 = 0;
F718 = 0;
F719 = 0;
F720 = 1/cx8*sx9*(omega_q_m-x(14));
F721 = 1/cx8*cx9*(omega_r_m-x(15));
F87 = 0;
F88 = 0;
F89 = -sx9*x(20)*(omega_q_m-x(14))-cx9*x(21)*(omega_r_m-x(15));
F810 = 0;
F811 = 0;
F812 = 0;
F813 = 0;
F814 = -cx9*x(20);
F815 = sx9*x(21);
F816 = 0;
F817 = 0;
F818 = 0;
F819 = 0;
F820 = cx9*(omega_q_m-x(14));
```

```
F821 = -sx9*(omega_r_m-x(15));
F97 = 0;
F98 = (1+tx8^2)*(sx9*x(20)*(omega_q_m-x(14))+cx9*x(21)*(omega_r_m-x(15)));
F99 = tx8*(cx9*x(20)*(omega_q_m-x(14))-sx9*x(21)*(omega_r_m-x(15)));
F910 = 0;
F911 = 0;
F912 = 0;
F913 = -x(19);
F914 = -tx8*sx9*x(20);
F915 = -tx8*cx9*x(21);
F916 = 0;
F917 = 0;
F918 = 0;
F919 = omega_p_m-x(13);
F920 = tx8*sx9*(omega_q_m-x(14));
F921 = tx8*cx9*(omega_r_m-x(15));

F = [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 0 F47 F48 F49 F410 F411 F412 F413 F414 F415 F416 F417 F418 F419 F420 F421
   0 0 0 0 0 0 F57 F58 F59 F510 F511 F512 F513 F514 F515 F516 F517 F518 F519 F520 F521
   0 0 0 0 0 0 F67 F68 F69 F610 F611 F612 F613 F614 F615 F616 F617 F618 F619 F620 F621
   0 0 0 0 0 0 F77 F78 F79 F710 F711 F712 F713 F714 F715 F716 F717 F718 F719 F720 F721
   0 0 0 0 0 0 F87 F88 F89 F810 F811 F812 F813 F814 F815 F816 F817 F818 F819 F820 F821
   0 0 0 0 0 0 F97 F98 F99 F910 F911 F912 F913 F914 F915 F916 F917 F918 F919 F920 F921
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];

st = length(F) + 1;

% Filling the P matrix with the states
for i=1:length(F)

   for j=1:length(F)

      P(i,j) = x(st);
```

```
        st = st + 1;

    end

end

% Variation of Ricatti equation
P_dot = F*P + P*F' + G*Q*G';

st = length(F) + 1;

% Filling the functions with the elements of P_dot
for i=1:length(F)

    for j=1:length(F)

        f(st) = P_dot(i,j);

        st = st + 1;

    end

end
```

APPENDIX B

C++ code that includes the definitions of variables, initialization parameters, and function calls to the KF, Runge Kutta and IMU/GPS communication programs (the MAIN program):

```
// Author:  Caleb Wells

// Main program

# include <cstdlib>
# include <iostream>
# include <iomanip>
# include <cmath>
#include <conio.h>
#include "Matrix.h"
#include "kalmanfilter.h"
#include "rungeKutta.h"
#include "MTGPS.h"
#include "IMUSerial.h"
#include "IPC.h"
//#define _CRTDBG_MAP_ALLOC
//#include <stdlib.h>
//#include <crtdbg.h>

using namespace std;

#ifndef PI
#define PI 3.1415926535
#endif
#ifndef IMU_Hz
#define IMU_Hz 71.1
#endif
#define R_EARTH 6366564.864
#define g 9.807

// Defining all the elements of the Process Noise Covariance Matrix
#define X_ACCEL_NOISE .01 * g
#define Y_ACCEL_NOISE .01 * g
#define Z_ACCEL_NOISE .01 * g
#define OMEGA_P_NOISE  .5 * PI / 180
#define OMEGA_Q_NOISE  .5 * PI / 180
#define OMEGA_R_NOISE  1.0 * PI / 180

// Beginning yaw angle of the IMU
#define YAW_BEG -40 * PI / 180

// Defining the initial offsets of the origin of the GPS antenna co-
ordinate system and the Earth co-ordinate system
#define N_OFF X_OFF*cos(YAW_BEG) - Y_OFF*sin(YAW_BEG)
#define E_OFF X_OFF*sin(YAW_BEG) + Y_OFF*cos(YAW_BEG)
#define D_OFF Z_OFF
```

```
// Yaw offset from IMU to centerline of AGV
#define YAW_OFF 0 * PI / 180

// Number of points to average while sitting to define the origin of
the ICS
#define NUM_AVG 20

extern Matrix* R_Mat;
extern Matrix* identity;
extern double state_est[21];

double x_accel_m;
double y_accel_m;
double z_accel_m;
double omega_p_m;
double omega_q_m;
double omega_r_m;
int fail=0;
int first_iter =0;
double state_est[21];
double array2[6];
double ORIGIN_LAT;
double ORIGIN_LON;
double ORIGIN_ALT;

int GPSFLAG;

/*FILES FOR LOGGING*/
FILE * kflog;
FILE * imulog;
FILE * rklog;


Matrix* Q;
Matrix* G;
Matrix* P;
Matrix* F;
Matrix* P_dot;
Matrix* P_est;

//typedef enum {init, transmission, header, set_reference,
clear_reference, binary, decimal} IMU_telegram;
int main ( void );

void RK_KF ();

void initialize();

void dump(void);
DWORD WINAPI GPS_Flagger(LPVOID Param);
void GPS_Stopper();

//*****************************************************************
*********
```

```
int main( void )
{
	double lat_sum = 0;
	double lon_sum = 0;
	double alt_sum = 0;
	//atexit(dump);
	//_CrtSetBreakAlloc(455855);

	imulog = fopen("imu_log.xls","w");
	kflog = fopen("kf_log.xls","w");
	rklog = fopen("rk_log.txt","w");
	if(!imulog||!kflog||!rklog)
	{
		printf("Error Opening Files");
		return 0;
	}
	//initialize
	initialize();
	GPSreading * GPSData = (GPSreading*)malloc(sizeof(GPSreading));
IMU_init();  //start the IMU
//Wait until the IMU starts sending the header, then...
wait_for_data();
//Wait 3 seconds
Sleep(3000);
//Send the first space bar "hit"
IMU((IMU_telegram)0);
//Wait a quarter of a second since back to back hits won't work
Sleep(250);
//Send another space bar "hit"
IMU((IMU_telegram)0);

//Wait for streaming data from the IMU
wait_for_data();
//Read 25 lines, just to let it get going...
for(int i = 0; i < 25; i++)
	read_IMU_line(array2);
//Then send the 'r' character to set it in reference mode.
IMU((IMU_telegram)3);



	//start threads
while(	startGPSSerial()==0)
{
	printf("Error Starting Thread");
}



	//KF
	double * y_t;
	y_t = (double*)malloc(sizeof(double)*3);
```

```
	//get first gps()  (GPS on separate thread

	// Setting up the origin of the GPS antenna
for(int i = 0; i<NUM_AVG;i++)
{
	while(getGPS(&GPSData)==0)
	{
		if(i==0)
			printf("w\n");
		Sleep(10);
	}
	lat_sum+=GPSData->lat;
	lon_sum+=GPSData->lon;
	alt_sum+=GPSData->alt;

}


	ORIGIN_LAT = lat_sum/NUM_AVG;
	ORIGIN_LON = lon_sum/NUM_AVG;
	ORIGIN_ALT = alt_sum/NUM_AVG;

	y_t[0] = N_OFF;
	y_t[1] = E_OFF;
	y_t[2] = D_OFF;

	kalmanFilter(y_t);


//loop
	while(1)
	{
		//BLOCK IMU (wait until you get an IMU)
		read_IMU_line(array2);  //how big is the IMU array, what is
the time thing
		//RK  for XX ms (1/Hz)

x_accel_m = array2[0];
y_accel_m = array2[1];
z_accel_m = array2[2];
omega_p_m = array2[3];
omega_q_m = array2[4];
omega_r_m = array2[5];

		// Modifying the incoming measurements from g's and deg/s
to m/s/s and rad/s
		x_accel_m *= g;
		y_accel_m *= g;
		z_accel_m *= g;
		omega_p_m *= PI/180.;
		omega_q_m *= PI/180.;
		omega_r_m *= PI/180.;
		if(imulog)
		{
```

```
      fprintf(imulog,"%lf\t%lf\t%lf\t%lf\t%lf\t%lf\n",x_accel_m/g,y_acc
el_m/g,z_accel_m/g,omega_p_m*180/PI,omega_q_m*180/PI,omega_r_m*180/PI);

      //printf("%lf\t%lf\t%lf\n",omega_p_m*180/PI,omega_q_m*180/PI,omeg
a_r_m*180/PI);
            }
            else
            {
                  printf("ERROR\n");
            }
            RK_KF();


            //CHECK for GPS
            //if gps
            //    run KF


            if(getGPS(&GPSData)==1)
            {
                  //set the values here
                  //printf("NEW GPS\n");
                  y_t[0] = 2*R_EARTH*sin(((GPSData->lat*PI/180) -
(ORIGIN_LAT * PI/180))/2) + N_OFF;
                  y_t[1] = 2*R_EARTH*sin(((GPSData->lon*PI/180) -
(ORIGIN_LON * PI/180))/2) + E_OFF;
                  y_t[2] = ORIGIN_ALT - GPSData->alt + D_OFF;
                  if(GPSFLAG==0)
                        kalmanFilter(y_t);


      fprintf(kflog,"%d\t%.9lf\t%.9lf\t%.9lf\t%.9lf\t%.9lf\t%.9lf\t%.9l
f\t%.9lf\t%.9lf\t%.9lf\t%.9lf\t%.9lf\t%.9lf\t%.9lf\t%.9lf\t%.9lf\t%.9lf
\t%.9lf\t%.9lf\t%.9lf\t%.9lf\t%.9lf\t%.9lf\t%.9lf\n",GPSFLAG,state_est[
0],state_est[1],state_est[2],state_est[3],state_est[4],state_est[5],sta
te_est[6]*180/PI,state_est[7]*180/PI,state_est[8]*180/PI,state_est[9]/g
,state_est[10]/g,state_est[11]/g,state_est[12]*180/PI,state_est[13]*180
/PI,state_est[14]*180/PI,state_est[15],state_est[16],state_est[17],stat
e_est[18],state_est[19],state_est[20],y_t[0],y_t[1],y_t[2],GPSData-
>numSat);
            }
//          printf("yaw = %lf\npitch = %lf\nroll = %lf\nN = %lf\nE =
%lf\nD =
%lf\n\n",state_est[6]*180/PI,state_est[7]*180/PI,state_est[8]*180/PI,st
ate_est[0],state_est[1],state_est[2]);
            if(quit==1)
                  break;

            char* cont_message = (char*)malloc(sizeof(char)*150);
            sprintf(cont_message,"%lf,%lf,%lf,%lf,%lf",GPSData-
>lat,GPSData->lon,state_est[6]+YAW_OFF,state_est[7],state_est[8]);
            sendMSG(cont_message,CLIENT);
            free(cont_message);
```

```
	}

}


void RK_KF()
{
# define NEQN 462



	int flag;

	double y[NEQN];


	flag = 1;


	first_iter = 1;

	int st = 0;
	for(int i = 0;i<21;i++)
	{
		y[st] = state_est[i];
		st++;
	}
	for(int i=0;i<21;i++)
		for(int j=0;j<21;j++)
		{
			y[st] = P_est->mat[i][j];
			st++;
		}

		rungeKutta(y,1./IMU_Hz,1./IMU_Hz,462);

		SYSTEMTIME sys_time;
		FILETIME new_time;
		GetSystemTime(&sys_time);
		SystemTimeToFileTime(&sys_time, &new_time);
		ULARGE_INTEGER                      *time =
(ULARGE_INTEGER*)(&new_time);


	fprintf(rklog,"%I64i\t%.9lf\t%.9lf\t%.9lf\t%.9lf\t%.9lf\t%.9lf\t%
.9lf\t%.9lf\t%.9lf\t%.9lf\t%.9lf\t%.9lf\t%.9lf\t%.9lf\t%.9lf\t%.
9lf\t%.9lf\t%.9lf\t%.9lf\t%.9lf\n",time-
>QuadPart,y[0],y[1],y[2],y[3],y[4],y[5],y[6]*180/PI,y[7]*180/PI,y[8]*18
0/PI,y[9]/g,y[10]/g,y[11]/g,y[12]*180/PI,y[13]*180/PI,y[14]*180/PI,y[15
],y[16],y[17],y[18],y[19],y[20]);

		for(st=0;st<21;st++)
		{
```

```
                state_est[st] = y[st];
        }


        for(int i=0;i<21;i++)
                for(int j=0;j<21;j++)
                {
                        P_est->mat[i][j] = y[st];
                        st++;
                }
                return;
# undef NEQN
}
//*****************************************************************
*********

void initialize()
{

        // Process Noise Covariance Matrix
        double Q_data[36] = {
                pow(X_ACCEL_NOISE,2),0,0,0,0,0,
                0,pow(Y_ACCEL_NOISE,2),0,0,0,0,
                0,0,pow(Z_ACCEL_NOISE,2),0,0,0,
                0,0,0,pow(OMEGA_P_NOISE,2),0,0,
                0,0,0,0,pow(OMEGA_Q_NOISE,2),0,
                0,0,0,0,0,pow(OMEGA_R_NOISE,2)};


        // Initial Estimation Error Covariance Matrix
        double P_est_data[441] = {
                5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                0,0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                0,0,0,0,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                0,0,0,0,0,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,
                0,0,0,0,0,0,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,
                0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,
                0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,
                0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,
                0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,
                0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,
                0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,
                0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,.05,0,0,0,0,0,
                0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,.05,0,0,0,0,
                0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,.05,0,0,0,
                0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,.05,0,0,
                0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,.05,0,
                0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,.05};

        // Identity Matrix
```

```
double iden_data[441] = {
      1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
      0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
      0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
      0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
      0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
      0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
      0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
      0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,
      0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,
      0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,
      0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,
      0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,
      0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,
      0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,
      0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,
      0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,
      0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,
      0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,
      0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
      0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,
      0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1};

// Measurement Noise Covariance Matrix
double R_data[9] = {
      .1,0,0,
      0,.1,0,
      0,0,.2};

Q = createMat(6,6);
identity = createMat(21,21);
G = createMat(21,6);
R_Mat = createMat(3,3);
P = createMat(21,21);
F = createMat(21,21);
P_dot = createMat(21,21);
P_est = createMat(21,21);
fillMat(identity,iden_data);
fillMat(Q,Q_data);
fillMat(P_est,P_est_data);
fillMat(R_Mat,R_data);

// Initial Estimates
for(int i=0;i<15;i++)
      state_est[i] = 0;
state_est[3] = 0;
state_est[4] = 0;
state_est[6] = YAW_BEG-YAW_OFF;
state_est[15] = 1;
state_est[16] = 1;
state_est[17] = 1;
state_est[18] = 1;
state_est[19] = 1;
state_est[20] = 1;
```

```
            GPSFLAG = 0;
            GPS_Stopper();
            Sleep(2000);
            printf("Setting Up Server\n");
            setupServer();   //setup the IPC Server
            printf("Connection Established\n");

            Sleep(200);
}

//void dump(void)
//{
//    _CrtDumpMemoryLeaks();
//}

void GPS_Stopper()
{
      DWORD ThreadID;
      HANDLE ThreadHandle;
      int Param;
      Param = 5;
      newRead = 0;

      //create the thread
      ThreadHandle = CreateThread(
            NULL,  //default security attributes
            0,          //default stack size
            GPS_Flagger,      //thread Function
            &Param,     //parameter to thread function
            0,          //default creation flags
            &ThreadID); //returns the thread Identifier


      return;
}

DWORD WINAPI GPS_Flagger(LPVOID Param)
{
      char test;

      while(1)
      {
            fflush(stdin);
            scanf("%c",&test);
            printf("\n");
            if(test == 'k' || test == 'K')
            {

                  if(GPSFLAG == 1)
                  {
                        printf("GPS Resumed\n");
                        GPSFLAG = 0;
                  }
```

```
                else
                {
                        printf("GPS Blocked\n");
                        GPSFLAG = 1;
                }
            }
            else
            {
                printf("Invalid Input\n");
            }
      }
return 0;
}
```

C++ code that includes the KF algorithm:

```
// Author:  Caleb Wells

// Kalman Filter algorithm code

#include <iostream>
#include <conio.h>
#include "Matrix.h"

using namespace std;

// Defining the offsets from the IMU to the GPS antenna
#define X_OFF -45.0/39.31
#define Y_OFF 6.0/39.31
#define Z_OFF -51.0/39.31

extern Matrix* P_est;
extern double state_est[21];


Matrix* R_Mat;
Matrix* identity;
//R_mat = fillMat(R_Mat,R_data);

double trigMatKF[9];  // matrix to store trig conversions (for KF)

//prototypes

void kalmanFilter(double * y_Tilde)
{

      double *h_data;
      Matrix* H_Mat = createMat(3,21);
      Matrix* h = createMat(3,1);
      Matrix* K = createMat(21,3);
      Matrix* y_T = createMat(3,1);
      Matrix* DCM_AP = createMat(3,3);
```

```
      Matrix* state = createMat(21,1);

      trigMatKF[0] = cos(state_est[6]);
      trigMatKF[1] = sin(state_est[6]);
      trigMatKF[2] = tan(state_est[6]);
      trigMatKF[3] = cos(state_est[7]);
      trigMatKF[4] = sin(state_est[7]);
      trigMatKF[5] = tan(state_est[7]);
      trigMatKF[6] = cos(state_est[8]);
      trigMatKF[7] = sin(state_est[8]);
      trigMatKF[8] = tan(state_est[8]);

      // Defining the Direction Cosine Matrix from the ICS to BCS
      double DCM_data[9] = {
                  trigMatKF[3]*trigMatKF[0],trigMatKF[3]*trigMatKF[1],-
trigMatKF[4],
                  trigMatKF[4]*trigMatKF[7]*trigMatKF[0]-
trigMatKF[6]*trigMatKF[1],trigMatKF[4]*trigMatKF[7]*trigMatKF[1]+trigMa
tKF[6]*trigMatKF[0],trigMatKF[3]*trigMatKF[7],

      trigMatKF[4]*trigMatKF[6]*trigMatKF[0]+trigMatKF[7]*trigMatKF[1],
trigMatKF[4]*trigMatKF[6]*trigMatKF[1]-
trigMatKF[7]*trigMatKF[0],trigMatKF[3]*trigMatKF[6]};

      fillMat(DCM_AP,DCM_data);
//    print(DCM_AP);

      Matrix* DCM_APt = trans(DCM_AP);
//    print(DCM_APt);

      h_data = (double *)malloc(3*sizeof(double));
      h_data[0] = state_est[0] + DCM_APt->mat[0][0]*X_OFF + DCM_APt-
>mat[0][1]*Y_OFF + DCM_APt->mat[0][2]*Z_OFF;
      h_data[1] = state_est[1] + DCM_APt->mat[1][0]*X_OFF + DCM_APt-
>mat[1][1]*Y_OFF + DCM_APt->mat[1][2]*Z_OFF;
      h_data[2] = state_est[2] + DCM_APt->mat[2][0]*X_OFF + DCM_APt-
>mat[2][1]*Y_OFF + DCM_APt->mat[2][2]*Z_OFF;

      double H06 = -trigMatKF[3]*trigMatKF[1]*X_OFF+(-
trigMatKF[4]*trigMatKF[7]*trigMatKF[1]-
trigMatKF[6]*trigMatKF[0])*Y_OFF+(-
trigMatKF[4]*trigMatKF[6]*trigMatKF[1]+trigMatKF[7]*trigMatKF[0])*Z_OFF
;
      double H07 = -
trigMatKF[4]*trigMatKF[0]*X_OFF+trigMatKF[3]*trigMatKF[7]*trigMatKF[0]*
Y_OFF+trigMatKF[3]*trigMatKF[6]*trigMatKF[0]*Z_OFF;
      double H08 =
(trigMatKF[4]*trigMatKF[6]*trigMatKF[0]+trigMatKF[7]*trigMatKF[1])*Y_OF
F+(-
trigMatKF[4]*trigMatKF[7]*trigMatKF[0]+trigMatKF[6]*trigMatKF[1])*Z_OFF
;
      double H16 =
trigMatKF[3]*trigMatKF[0]*X_OFF+(trigMatKF[4]*trigMatKF[7]*trigMatKF[0]
-
```

```
trigMatKF[6]*trigMatKF[1])*Y_OFF+(trigMatKF[4]*trigMatKF[6]*trigMatKF[0
]+trigMatKF[7]*trigMatKF[1])*Z_OFF;
      double H17 = -
trigMatKF[4]*trigMatKF[1]*X_OFF+trigMatKF[3]*trigMatKF[7]*trigMatKF[1]*
Y_OFF+trigMatKF[3]*trigMatKF[6]*trigMatKF[1]*Z_OFF;
      double H18 = (trigMatKF[4]*trigMatKF[6]*trigMatKF[1]-
trigMatKF[7]*trigMatKF[0])*Y_OFF+(-
trigMatKF[4]*trigMatKF[7]*trigMatKF[1]-
trigMatKF[6]*trigMatKF[0])*Z_OFF;
      double H26 = 0;
      double H27 = -trigMatKF[3]*X_OFF-trigMatKF[4]*trigMatKF[7]*Y_OFF-
trigMatKF[4]*trigMatKF[6]*Z_OFF;
      double H28 = trigMatKF[3]*trigMatKF[6]*Y_OFF-
trigMatKF[3]*trigMatKF[7]*Z_OFF;

      // Measurement Sensitivity Matrix
      double H_Mat_data[63] = {
                  1,0,0,0,0,0,H06,H07,H08,0,0,0,0,0,0,0,0,0,0,0,0,
                  0,1,0,0,0,0,H16,H17,H18,0,0,0,0,0,0,0,0,0,0,0,0,
                  0,0,1,0,0,0,H26,H27,H28,0,0,0,0,0,0,0,0,0,0,0,0};

      fillMat(H_Mat,H_Mat_data);
      fillMat(h,h_data);
      fillMat(y_T,y_Tilde);
//    print(H_Mat);
//    print(h);
//    print(y_T);


//    print(identity);



      fillMat(state,state_est);
//    print(state);

      Matrix* H_trans = trans(H_Mat);
      Matrix* temp1 = mul(P_est,H_trans);
      Matrix* temp2 = mul(H_Mat,P_est);
      Matrix* temp3 = mul(temp2,H_trans);
      Matrix* temp4 = add(temp3,R_Mat);
      Matrix* tempInv = inv(temp4);
      deleteMat(&K);  //save the memory before it gets reallocated
      K = mul(temp1,tempInv);

      deleteMat(&H_trans);
      deleteMat(&temp1);
      deleteMat(&temp2);
      deleteMat(&temp3);
      deleteMat(&temp4);
      deleteMat(&tempInv);

      Matrix * suba = sub(y_T,h);
      Matrix * multa = mul(K,suba);
```

```
	Matrix * tempa = add(state, multa);

	deleteMat(&suba);
	deleteMat(&multa);

	deleteMat(&state);  //save the memory before it gets reallocated
	state = tempa;
	Matrix * mult = mul(K,H_Mat);
	Matrix * subt = sub(identity,mult);
	Matrix * tempa2 =  mul( subt, P_est);
	deleteMat(&mult);
	deleteMat(&subt);

	deleteMat(&P_est);  //save the memory of P_est
	P_est = tempa2;

	for(int i = 0; i<21; i++)
	{
		state_est[i] = state->mat[i][0];
	}


	deleteMat(&h);
	deleteMat(&DCM_AP);
	deleteMat(&DCM_APt);
	deleteMat(&K);
	deleteMat(&y_T);
	deleteMat(&state);
	deleteMat(&H_Mat);
	if(h_data)
	{
		free(h_data);
		h_data = NULL;
	}
}
```

C++ code that runs Runge-Kutta 4th order to produce the *apriori* estimates (including the estimation error covariance matrix):

```
// Author:  Unknown, but modified by Caleb Wells

// Runge Kutta 4th order code

/* Runge Kutta for a set of first order differential equations */

#include "stdAfx.h"
#include "PreciseTimer.h"

#include <iostream>
#include <string>
#include <algorithm>
```

```
using namespace std;
#include "Matrix.h"
#include <math.h>

using namespace std;

extern Matrix* Q;
extern Matrix* G;
extern Matrix* P;
extern Matrix* F;
extern Matrix* P_dot;
extern Matrix* P_est;

extern double x_accel_m;
extern double y_accel_m;
extern double z_accel_m;
extern double omega_p_m;
extern double omega_q_m;
extern double omega_r_m;

#define N 462                       /* number of first order equations */
#ifndef PI
//#define PI     4.0*atan(1.0)
#define PI 3.1415926535
#endif
#ifndef IMU_Hz
#define IMU_Hz 71.1
#endif
#define g 9.807

double trigMat[9];            //matrix to store trig conversions
FILE *output;                   /* internal filename */
void rungeKutta(double y[],double timestep,double maxTime,int NUMEQ);
void runge4(double y[], double step); /* Runge-Kutta function */

void f(double y[]);          /* function for derivatives */
double yp[N];

void rungeKutta(double y[],double timestep,double maxTime,int NUMEQ)
{
      double t;
      int j;
      for (j=1; j*timestep<=maxTime ;j++)                    /* time
loop */
      {
            t=j*timestep;
            runge4(y, timestep);

      }
}

void runge4(double y[], double step)
{
```

```
      double h=step/2.0,                        /* the midpoint */
            t1[N], t2[N], t3[N],              /* temporary storage arrays
*/
            k1[N], k2[N], k3[N],k4[N];       /* for Runge-Kutta */
      int i;

      f(y);

      for (i=0;i<N;i++)
            t1[i]=y[i]+0.5*(k1[i]=step*yp[i]);

      //    CPreciseTimer timer;
      //    timer.StartTimer();


      f(t1);

      for (i=0;i<N;i++)
            t2[i]=y[i]+0.5*(k2[i]=step*yp[i]);

      f(t2);

      for (i=0;i<N;i++)
            t3[i]=y[i]+ (k3[i]=step*yp[i]);

      f(t3);

      for (i=0;i<N;i++)
            k4[i]= step*yp[i];
      for (i=0;i<N;i++)
            y[i]+=(k1[i]+2*k2[i]+2*k3[i]+k4[i])/6.0;
      //    timer.StopTimer();

      //    __int64 i64Diff = timer.GetTime();
      //    printf("Diff2 (PreciseTimer) =
%lf\n",(double)i64Diff*.000001);

}

void  f(double y[])
{
      int st;

      double
f3x,f3y,f3z,f4x,f4y,f4z,f5x,f5y,f5z,a_x,a_y,a_z,w_p,w_q,w_r;
      double
F36,F37,F38,F39,F310,F311,F312,F313,F314,F315,F316,F317,F318,F319,F320;
    double
F46,F47,F48,F49,F410,F411,F412,F413,F414,F415,F416,F417,F418,F419,F420;
      double
F56,F57,F58,F59,F510,F511,F512,F513,F514,F515,F516,F517,F518,F519,F520;
      double
F66,F67,F68,F69,F610,F611,F612,F613,F614,F615,F616,F617,F618,F619,F620;
```

```
      double
F76,F77,F78,F79,F710,F711,F712,F713,F714,F715,F716,F717,F718,F719,F720;
      double
F86,F87,F88,F89,F810,F811,F812,F813,F814,F815,F816,F817,F818,F819,F820;

      // Truth model
      w_p = y[18]*(omega_p_m - y[12]);
      w_q = y[19]*(omega_q_m - y[13]);
      w_r = y[20]*(omega_r_m - y[14]);

      trigMat[0] = cos(y[6]);
      trigMat[1] = sin(y[6]);
      trigMat[2] = tan(y[6]);
      trigMat[3] = cos(y[7]);
      trigMat[4] = sin(y[7]);
      trigMat[5] = tan(y[7]);
      trigMat[6] = cos(y[8]);
      trigMat[7] = sin(y[8]);
      trigMat[8] = tan(y[8]);

      f3x = trigMat[0]*trigMat[3];
      f3y = trigMat[7]*trigMat[4]*trigMat[0]-trigMat[6]*trigMat[1];
      f3z = trigMat[6]*trigMat[4]*trigMat[0]+trigMat[7]*trigMat[1];
      f4x = trigMat[1]*trigMat[3];
      f4y = trigMat[7]*trigMat[4]*trigMat[1]+trigMat[6]*trigMat[0];
      f4z = trigMat[6]*trigMat[4]*trigMat[1]-trigMat[7]*trigMat[0];
      f5x = -trigMat[4];
      f5y = trigMat[7]*trigMat[3];
      f5z = trigMat[6]*trigMat[3];

      a_x = y[15]*(x_accel_m - y[9]) - g*trigMat[4];
      a_y = y[16]*(y_accel_m - y[10]) + g*trigMat[3]*trigMat[7];
      a_z = y[17]*(z_accel_m - y[11]) + g*trigMat[3]*trigMat[6];

      // Equations of motion that are integrated by RK4
      yp[0] = y[3];
      yp[1] = y[4];
      yp[2] = y[5];
      yp[3] = f3x*a_x + f3y*a_y + f3z*a_z;
      yp[4] = f4x*a_x + f4y*a_y + f4z*a_z;
      yp[5] = f5x*a_x + f5y*a_y + f5z*a_z;
      yp[6] = 1/trigMat[3]*(trigMat[7]*w_q+trigMat[6]*w_r);
      yp[7] = trigMat[6]*w_q - trigMat[7]*w_r;
      yp[8] = w_p + trigMat[5]*(trigMat[7]*w_q+trigMat[6]*w_r);
      yp[9] = 0;
      yp[10] = 0;
      yp[11] = 0;
      yp[12] = 0;
      yp[13] = 0;
      yp[14] = 0;
      yp[15] = 0;
      yp[16] = 0;
      yp[17] = 0;
      yp[18] = 0;
```

```
      yp[19] = 0;
      yp[20] = 0;

      // Process Noise Mapping Matrix
      double G_data[126] = {
                  0,0,0,0,0,0,
                  0,0,0,0,0,0,
                  0,0,0,0,0,0,
                  f3x*y[15],f3y*y[16],f3z*y[17],0,0,0,
                  f4x*y[15],f4y*y[16],f4z*y[17],0,0,0,
                  f5x*y[15],f5y*y[16],f5z*y[17],0,0,0,

      0,0,0,0,1/trigMat[3]*trigMat[7]*y[19],1/trigMat[3]*trigMat[6]*y[2
0],
                  0,0,0,0,trigMat[6]*y[19],-trigMat[7]*y[20],

      0,0,0,y[18],trigMat[5]*trigMat[7]*y[19],trigMat[5]*trigMat[6]*y[2
0],
                  0,0,0,0,0,0,
                  0,0,0,0,0,0,
                  0,0,0,0,0,0,
                  0,0,0,0,0,0,
                  0,0,0,0,0,0,
                  0,0,0,0,0,0,
                  0,0,0,0,0,0,
                  0,0,0,0,0,0,
                  0,0,0,0,0,0,
                  0,0,0,0,0,0,
                  0,0,0,0,0,0};

      fillMat(G,G_data);

      // Matrix of Partial Derivatives
      F36 = -trigMat[1]*trigMat[3]*(y[15]*(x_accel_m-y[9])-
g*trigMat[4])+(-trigMat[7]*trigMat[4]*trigMat[1]-
trigMat[6]*trigMat[0])*(y[16]*(y_accel_m-
y[10])+g*trigMat[3]*trigMat[7])+(-
trigMat[6]*trigMat[4]*trigMat[1]+trigMat[7]*trigMat[0])*(y[17]*(z_accel
_m-y[11])+g*trigMat[3]*trigMat[6]);
      F37 = -trigMat[0]*trigMat[4]*(y[15]*(x_accel_m-y[9])-
g*trigMat[4])-
trigMat[0]*pow(trigMat[3],2)*g+trigMat[7]*trigMat[3]*trigMat[0]*(y[16]*
(y_accel_m-y[10])+g*trigMat[3]*trigMat[7])-
(trigMat[7]*trigMat[4]*trigMat[0]-
trigMat[6]*trigMat[1])*g*trigMat[4]*trigMat[7]+trigMat[6]*trigMat[3]*tr
igMat[0]*(y[17]*(z_accel_m-y[11])+g*trigMat[3]*trigMat[6])-
(trigMat[6]*trigMat[4]*trigMat[0]+trigMat[7]*trigMat[1])*g*trigMat[4]*t
rigMat[6];
      F38 =
(trigMat[6]*trigMat[4]*trigMat[0]+trigMat[7]*trigMat[1])*(y[16]*(y_acce
l_m-y[10])+g*trigMat[3]*trigMat[7])+(trigMat[7]*trigMat[4]*trigMat[0]-
trigMat[6]*trigMat[1])*g*trigMat[3]*trigMat[6]+(-
trigMat[7]*trigMat[4]*trigMat[0]+trigMat[6]*trigMat[1])*(y[17]*(z_accel
```

```
_m-y[11])+g*trigMat[3]*trigMat[6])-
(trigMat[6]*trigMat[4]*trigMat[0]+trigMat[7]*trigMat[1])*g*trigMat[3]*t
rigMat[7];
      F39 = -trigMat[0]*trigMat[3]*y[15];
      F310 = -(trigMat[7]*trigMat[4]*trigMat[0]-
trigMat[6]*trigMat[1])*y[16];
      F311 = -
(trigMat[6]*trigMat[4]*trigMat[0]+trigMat[7]*trigMat[1])*y[17];
      F312 = 0;
      F313 = 0;
      F314 = 0;
      F315 = trigMat[0]*trigMat[3]*(x_accel_m-y[9]);
      F316 = (trigMat[7]*trigMat[4]*trigMat[0]-
trigMat[6]*trigMat[1])*(y_accel_m-y[10]);
      F317 =
(trigMat[6]*trigMat[4]*trigMat[0]+trigMat[7]*trigMat[1])*(z_accel_m-
y[11]);
      F318 = 0;
      F319 = 0;
      F320 = 0;
      F46 = trigMat[0]*trigMat[3]*(y[15]*(x_accel_m-y[9])-
g*trigMat[4])+(trigMat[7]*trigMat[4]*trigMat[0]-
trigMat[6]*trigMat[1])*(y[16]*(y_accel_m-
y[10])+g*trigMat[3]*trigMat[7])+(trigMat[6]*trigMat[4]*trigMat[0]+trigM
at[7]*trigMat[1])*(y[17]*(z_accel_m-y[11])+g*trigMat[3]*trigMat[6]);
      F47 = -trigMat[1]*trigMat[4]*(y[15]*(x_accel_m-y[9])-
g*trigMat[4])-
trigMat[1]*pow(trigMat[3],2)*g+trigMat[7]*trigMat[3]*trigMat[1]*(y[16]*
(y_accel_m-y[10])+g*trigMat[3]*trigMat[7])-
(trigMat[7]*trigMat[4]*trigMat[1]+trigMat[6]*trigMat[0])*g*trigMat[4]*t
rigMat[7]+trigMat[6]*trigMat[3]*trigMat[1]*(y[17]*(z_accel_m-
y[11])+g*trigMat[3]*trigMat[6])-(trigMat[6]*trigMat[4]*trigMat[1]-
trigMat[7]*trigMat[0])*g*trigMat[4]*trigMat[6];
      F48 = (trigMat[6]*trigMat[4]*trigMat[1]-
trigMat[7]*trigMat[0])*(y[16]*(y_accel_m-
y[10])+g*trigMat[3]*trigMat[7])+(trigMat[7]*trigMat[4]*trigMat[1]+trigM
at[6]*trigMat[0])*g*trigMat[3]*trigMat[6]+(-
trigMat[7]*trigMat[4]*trigMat[1]-
trigMat[6]*trigMat[0])*(y[17]*(z_accel_m-
y[11])+g*trigMat[3]*trigMat[6])-(trigMat[6]*trigMat[4]*trigMat[1]-
trigMat[7]*trigMat[0])*g*trigMat[3]*trigMat[7];
      F49 = -trigMat[1]*trigMat[3]*y[15];
      F410 = -
(trigMat[7]*trigMat[4]*trigMat[1]+trigMat[6]*trigMat[0])*y[16];
      F411 = -(trigMat[6]*trigMat[4]*trigMat[1]-
trigMat[7]*trigMat[0])*y[17];
      F412 = 0;
      F413 = 0;
      F414 = 0;
      F415 = trigMat[1]*trigMat[3]*(x_accel_m-y[9]);
      F416 =
(trigMat[7]*trigMat[4]*trigMat[1]+trigMat[6]*trigMat[0])*(y_accel_m-
y[10]);
```

```
      F417 = (trigMat[6]*trigMat[4]*trigMat[1]-
trigMat[7]*trigMat[0])*(z_accel_m-y[11]);
      F418 = 0;
      F419 = 0;
      F420 = 0;
      F56 = 0;
      F57 = -trigMat[3]*(y[15]*(x_accel_m-y[9])-
g*trigMat[4])+trigMat[4]*g*trigMat[3]-
trigMat[7]*trigMat[4]*(y[16]*(y_accel_m-
y[10])+g*trigMat[3]*trigMat[7])-
pow(trigMat[7],2)*trigMat[3]*g*trigMat[4]-
trigMat[6]*trigMat[4]*(y[17]*(z_accel_m-
y[11])+g*trigMat[3]*trigMat[6])-
pow(trigMat[6],2)*trigMat[3]*g*trigMat[4];
      F58 = trigMat[6]*trigMat[3]*(y[16]*(y_accel_m-
y[10])+g*trigMat[3]*trigMat[7])-
trigMat[7]*trigMat[3]*(y[17]*(z_accel_m-
y[11])+g*trigMat[3]*trigMat[6]);
      F59 = trigMat[4]*y[15];
      F510 = -trigMat[7]*trigMat[3]*y[16];
      F511 = -trigMat[6]*trigMat[3]*y[17];
      F512 = 0;
      F513 = 0;
      F514 = 0;
      F515 = -trigMat[4]*(x_accel_m-y[9]);
      F516 = trigMat[7]*trigMat[3]*(y_accel_m-y[10]);
      F517 = trigMat[6]*trigMat[3]*(z_accel_m-y[11]);
      F518 = 0;
      F519 = 0;
      F520 = 0;
      F66 = 0;
      F67 = 1/pow(trigMat[3],2)*(trigMat[7]*y[19]*(omega_q_m-
y[13])+trigMat[6]*y[20]*(omega_r_m-y[14]))*trigMat[4];
      F68 = 1/trigMat[3]*(trigMat[6]*y[19]*(omega_q_m-y[13])-
trigMat[7]*y[20]*(omega_r_m-y[14]));
      F69 = 0;
      F610 = 0;
      F611 = 0;
      F612 = 0;
      F613 = -1/trigMat[3]*trigMat[7]*y[19];
      F614 = -1/trigMat[3]*trigMat[6]*y[20];
      F615 = 0;
      F616 = 0;
      F617 = 0;
      F618 = 0;
      F619 = 1/trigMat[3]*trigMat[7]*(omega_q_m-y[13]);
      F620 = 1/trigMat[3]*trigMat[6]*(omega_r_m-y[14]);
      F76 = 0;
      F77 = 0;
      F78 = -trigMat[7]*y[19]*(omega_q_m-y[13])-
trigMat[6]*y[20]*(omega_r_m-y[14]);
      F79 = 0;
      F710 = 0;
      F711 = 0;
```

```
      F712 = 0;
      F713 = -trigMat[6]*y[19];
      F714 = trigMat[7]*y[20];
      F715 = 0;
      F716 = 0;
      F717 = 0;
      F718 = 0;
      F719 = trigMat[6]*(omega_q_m-y[13]);
      F720 = -trigMat[7]*(omega_r_m-y[14]);
      F86 = 0;
      F87 = (1+pow(trigMat[5],2))*(trigMat[7]*y[19]*(omega_q_m-
y[13])+trigMat[6]*y[20]*(omega_r_m-y[14]));
      F88 = trigMat[5]*(trigMat[6]*y[19]*(omega_q_m-y[13])-
trigMat[7]*y[20]*(omega_r_m-y[14]));
      F89 = 0;
      F810 = 0;
      F811 = 0;
      F812 = -y[18];
      F813 = -trigMat[5]*trigMat[7]*y[19];
      F814 = -trigMat[5]*trigMat[6]*y[20];
      F815 = 0;
      F816 = 0;
      F817 = 0;
      F818 = omega_p_m-y[12];
      F819 = trigMat[5]*trigMat[7]*(omega_q_m-y[13]);
      F820 = trigMat[5]*trigMat[6]*(omega_r_m-y[14]);

      double F_data[441] = {
                  0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                  0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                  0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,

      0,0,0,0,0,0,F36,F37,F38,F39,F310,F311,F312,F313,F314,F315,F316,F3
17,F318,F319,F320,

      0,0,0,0,0,0,F46,F47,F48,F49,F410,F411,F412,F413,F414,F415,F416,F4
17,F418,F419,F420,

      0,0,0,0,0,0,F56,F57,F58,F59,F510,F511,F512,F513,F514,F515,F516,F5
17,F518,F519,F520,

      0,0,0,0,0,0,F66,F67,F68,F69,F610,F611,F612,F613,F614,F615,F616,F6
17,F618,F619,F620,

      0,0,0,0,0,0,F76,F77,F78,F79,F710,F711,F712,F713,F714,F715,F716,F7
17,F718,F719,F720,

      0,0,0,0,0,0,F86,F87,F88,F89,F810,F811,F812,F813,F814,F815,F816,F8
17,F818,F819,F820,
                  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
```

```
                  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

      fillMat(F,F_data);

      st = 21;
      for(int i=0;i<21;i++)
            for(int j=0;j<21;j++)
            {
                  P->mat[i][j] = y[st];
                  st++;
            }



            //          CPreciseTimer timer2;
            //          timer2.StartTimer();
            //          cout << endl << endl;

Matrix * F_Trans = trans(F);
Matrix * temp1 = mul(F,P);
Matrix * temp2 = mul(P,F_Trans);
Matrix * temp3 = mul(G,Q);
Matrix * temp4 = trans(G);
Matrix * temp5 = mul(temp3,temp4);
Matrix * add1 = add(temp1,temp2);

deleteMat(&P_dot);
P_dot = add( add1, temp5);

deleteMat(&F_Trans);
deleteMat(&add1);
deleteMat(&temp1);
deleteMat(&temp2);
deleteMat(&temp3);
deleteMat(&temp4);
deleteMat(&temp5);
            //          timer2.StopTimer();
            //    printf("%d\n",(int)timer2.SupportsHighResCounter());
            //          printf("Mat Mult =
%lf\n",(double)timer2.GetTime()*.000001);
            st = 21;
            for(int i = 0; i<21;i++)
                  for(int j = 0; j<21;j++)
                  {
                        yp[st] = P_dot->mat[i][j];
                        st++;
                  }
}
```

APPENDIX C

In order for the AGV to follow waypoints through a rich environment of obstacles, it must know the locations of those obstacles. This will be referred to as the mapping of obstacles. The SICK LMS [6] is the only source of terrain information for the TEES AGV. It pulses a laser at a rotating mirror to produce a 180° planar scan of the terrain. If the beam reflects off an object (a hit) the return is received by the SICK. The range is computed by measuring the time of flight of the pulsed beam. An encoder on the mirror provides the angle at which the hit was received. This information, contained in a single SICK 'scan', would be considered a local map of obstacles, that is, local to the current SICK co-ordinate system. However, for longer term use, i.e., in terms of minutes or so, it is necessary to store the locations of all recent as well as current obstacles, at least temporarily, which requires a 'global' map instead of a local map.

The global map is the collection of locations for all objects in the environment that have been sensed (recently) by the SICK and processed. These locations are in the ICS. This co-ordinate system has a stationary origin and axes. In order to map a SICK hit to the ICS, a detailed analysis must first be performed.

First, a more thorough understanding of the co-ordinate systems is required. Refer to figure 69 that contains the three co-ordinate systems: SICK, IMU and Inertial. Figures 69 through 73 were produced by Justin T. Bozalina in the Computer Science Department at Texas A&M University.

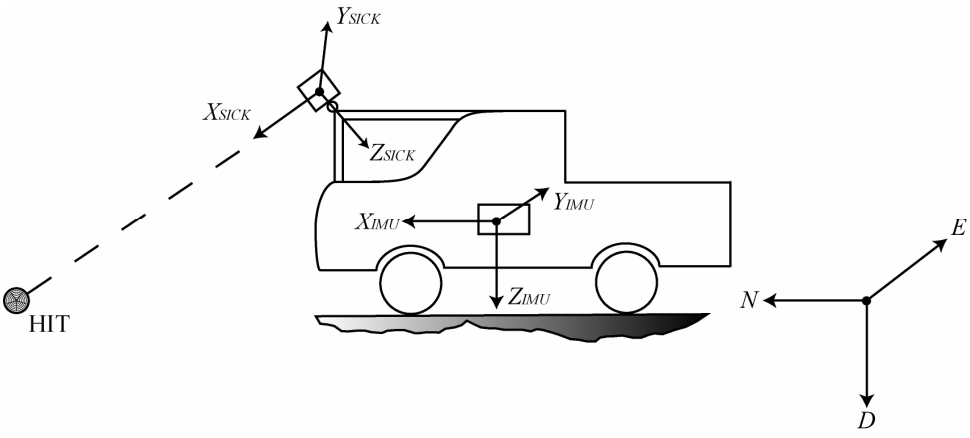

Figure 69. SICK, IMU and inertial co-ordinate systems on the AGV

1) Inertial Co-ordinate System – Also known as the Earth or Global Co-ordinate System. The axes and origin are as follows:

a. N - positive pointing towards North Pole

b. E - positive pointing 90° clockwise from North Pole

c. D - positive pointing towards center of Earth

d. Origin is designated by team

2) IMU Co-ordinate System – The body-fixed co-ordinate system attached to the IMU, which is (assumed) rigidly mounted to vehicle. Note that this co-ordinate system defines the orientation of the AGV because it is assumed that the AGV is a rigid body. The axes and origin are the following:

a. x_{IMU} - positive from rear of vehicle to front of vehicle, orthogonal to track

b. y_{IMU} - positive from driver's side of vehicle to passenger's side of vehicle, orthogonal to wheelbase

c. z_{IMU} - positive from roof of vehicle to undercarriage of vehicle, orthogonal to other IMU body co-ordinate axes

d. Origin is the center of the cluster of IMU sensors

3) SICK Co-ordinate System – The body-fixed co-ordinate system attached to the SICK LMS. The axes and origin are as follows:

a. x_{SICK} - positive from flat rear mount of SICK to front of SICK, orthogonal to flat rear mount

b. y_{SICK} - positive from left side of SICK to right side of SICK, looking from the rear of the SICK, parallel to y_{IMU}

c. z_{SICK} - positive from top of SICK to bottom of SICK, orthogonal to other SICK body co-ordinate axes

d. Origin is center of mirror located inside SICK

Refer again to figure 69 to gain additional understanding of the three co-ordinate systems. The next step is to provide the foundation for the order of rotations. It is essential that the different programs on the AGV use the same order of rotations. It was demonstrated earlier that the order of rotations for the AGV from ICS to BCS was yaw, pitch and then roll. These are

the first three rotations. The last rotation is the sweep of the SICK. The SICK LMS will also be rotated about its y-axis to provide a 3-D sensing of the environment. This movement will create the last rotation, the sweep, or the vertical angle, of the SICK. To eliminate any ambiguities or confusion, the definitions of each of the rotations will be discussed. Please refer to figures 70 through 73 that graphically show each of the rotations.

1) Yaw – Denoted as ψ, is the rotation about the D axis from the N axis to the projection of the x_{IMU} axis onto the $N-E$ plane
2) Pitch – Denoted as θ, is the rotation about the E' axis from the N' axis to the x_{IMU} axis
3) Roll – Denoted as ϕ, is the rotation about the N'' axis from the E'' axis to the y_{IMU} axis
4) Sweep – Denoted as α, is the rotation about the y_{IMU} axis from the x_{IMU} axis to the x_{SICK} axis

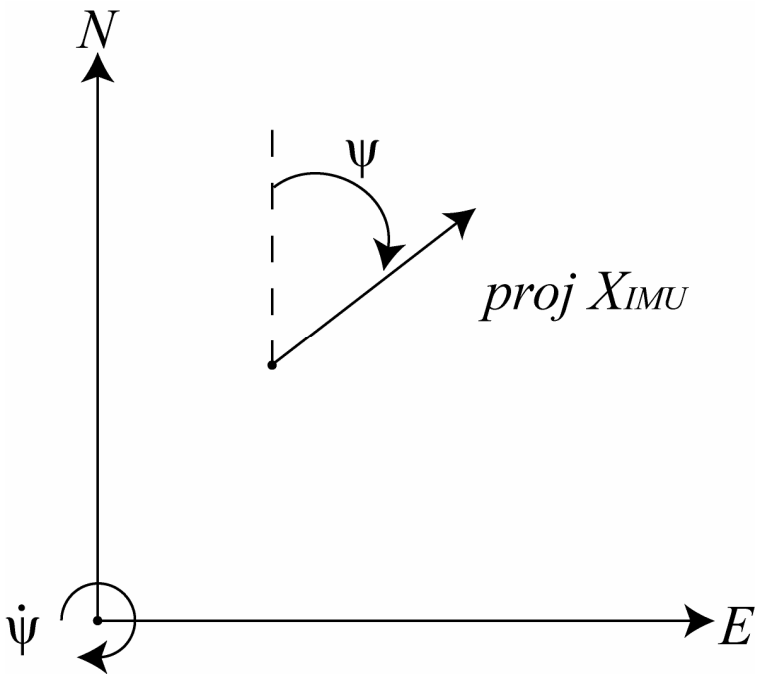

Figure 70. Yaw angle rotation

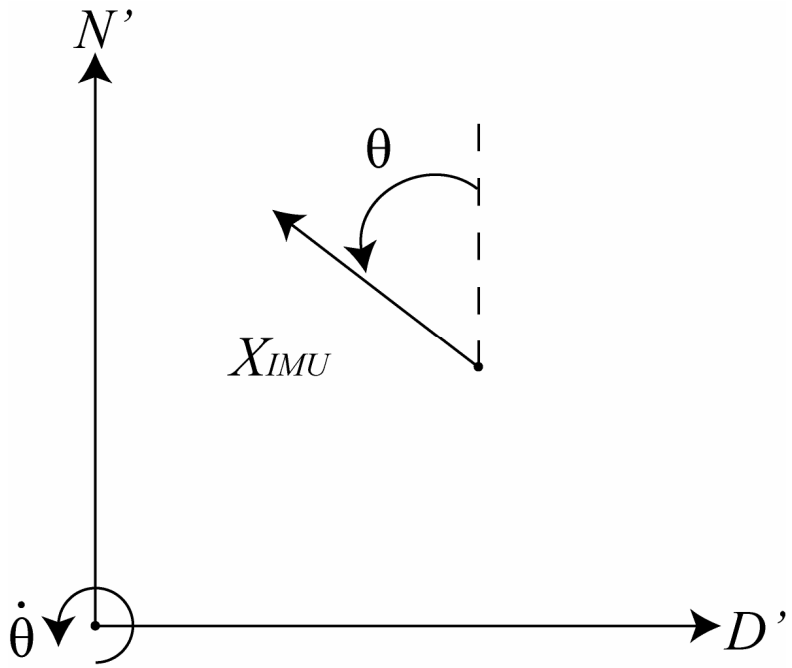

Figure 71. Pitch angle rotation

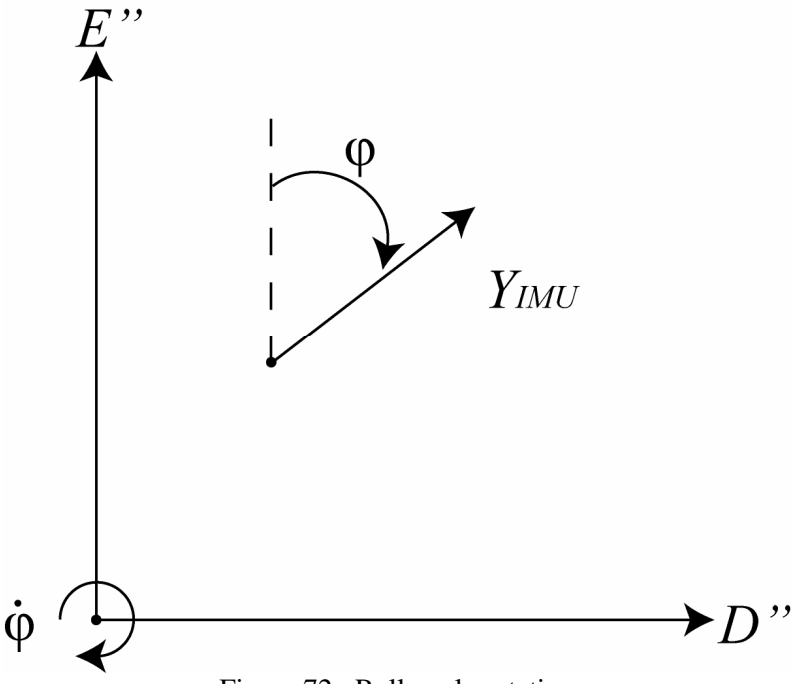

Figure 72. Roll angle rotation

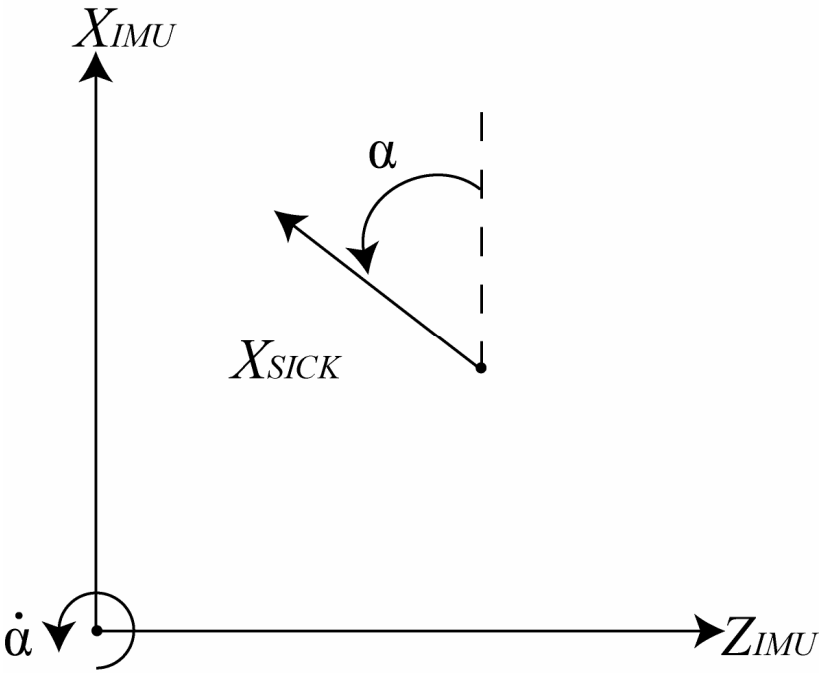

Figure 73. Sweep angle rotation

In order to continue with the mapping process, please refer to the nomenclature section regarding the following vector analysis. Figure 74 displays the vectors used to find the correct location of the SICK hit in the ICS.

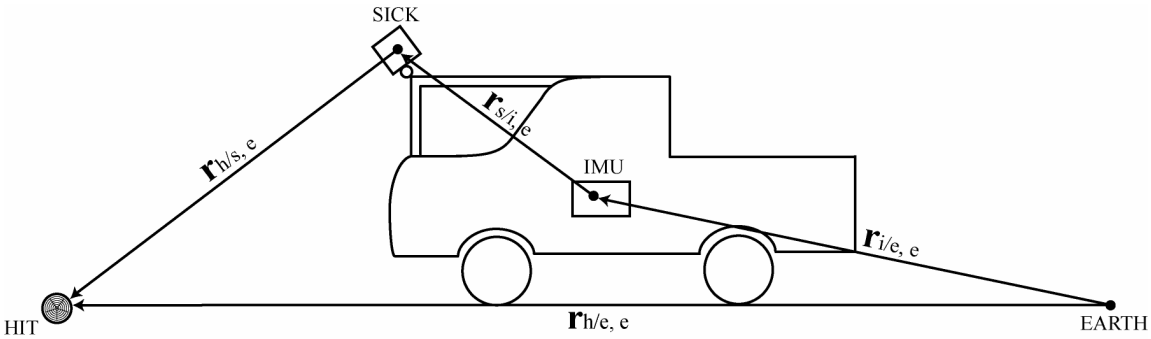

Figure 74. Vectors used to map the SICK hit to the ICS

When using vector analysis, there is some accounting that is required. One needs to follow a very systematic analysis and be consistent with notation in order that complete understanding of the material is achieved. Vector analysis is not the unique path to the solution, but it is a more graphical analysis which should be easier to understand.

For this mapping process, there are four steps:

1) Find the vector, in Earth co-ordinates, that locates the origin of the IMU co-ordinate system with respect to the origin of the ICS

$$\left(\vec{r}_{imu/earth}\right)_{earth} = \vec{r}_{i/e,e}$$

2) Find the vector, in Earth co-ordinates, that locates the origin of the SICK co-ordinate system with respect to the origin of the IMU co-ordinate system

$$\left(\vec{r}_{sick/imu}\right)_{earth} = \vec{r}_{s/i,e}$$

3) Find the vector, in Earth co-ordinates, that locates the SICK hit with respect to the origin of the SICK co-ordinate system

$$\left(\vec{r}_{hit/sick}\right)_{earth} = \vec{r}_{h/s,e}$$

4) Finally, add the above three vectors to find the vector, in Earth co-ordinates, that locates the SICK hit with respect to the origin of the ICS

$$\left(\vec{r}_{hit/earth}\right)_{earth} = \vec{r}_{h/e,e} = \vec{r}_{i/e,e} + \vec{r}_{s/i,e} + \vec{r}_{h/s,e}$$

The origin of the IMU co-ordinate system, in Earth co-ordinates, will be given by the KF. Therefore, part one is concluded. The origin of the SICK does not translate when a sweep occurs. Therefore $\vec{r}_{s/i,e}$ is dependent on the yaw, pitch and roll of the vehicle, along with $\vec{r}_{s/i,i}$. The rotation matrices below map ***forward*** (based on figures 70 through 73). These would be directly applied if the desire is to map the vector components in an inertial co-ordinate system to components in a body-fixed co-ordinate system. The objective here is the reverse or ***backward*** mapping. This requires the transpose of the forward rotation matrices, as well as reversing the order in which they are multiplied.

Therefore, the forward rotation matrices for yaw, pitch and roll are the following:

$$A_{yaw} = \begin{bmatrix} c\psi & s\psi & 0 \\ -s\psi & c\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$
$$A_{pitch} = \begin{bmatrix} c\theta & 0 & -s\theta \\ 0 & 1 & 0 \\ s\theta & 0 & c\theta \end{bmatrix} \quad \text{(C.1)}$$
$$A_{roll} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\phi & s\phi \\ 0 & -s\phi & c\phi \end{bmatrix}$$

Where $c\psi$ represents the cosine of ψ and $s\psi$ represents the sine of ψ. The same is true for θ and ϕ.

Therefore, $\vec{r}_{s/i,e}$ is the following:

$$\vec{r}_{s/i,e} = A_{yaw}^T A_{pitch}^T A_{roll}^T \vec{r}_{s/i,i} \quad \text{(C.2)}$$

The next task is to find $\vec{r}_{h/s,e}$. The forward rotation matrix for sweep is the following, along with the range vector:

$$A_{sweep} = \begin{bmatrix} c\alpha & 0 & -s\alpha \\ 0 & 1 & 0 \\ s\alpha & 0 & c\alpha \end{bmatrix}, \quad \vec{r}_{h/s,s} = \begin{bmatrix} m\sin\beta \\ m\cos\beta \\ 0 \end{bmatrix} \quad \text{(C.3)}$$

Where: m, β are the range and angle to the hit from the SICK in SICK co-ordinates, respectively (information provided by SICK)

Therefore, $\vec{r}_{h/s,e}$ is the following:

$$\vec{r}_{h/s,e} = A_{yaw}^T A_{pitch}^T A_{roll}^T A_{sweep}^T \vec{r}_{h/s,s} \quad \text{(C.4)}$$

The final task is to combine the previous three vectors to obtain the final vector, $\vec{r}_{h/e,e}$. This yields the following:

$$\vec{r}_{h/e,e} = \vec{r}_{h/s,e} + \vec{r}_{s/i,e} + \vec{r}_{i/e,e}$$

$$\Downarrow$$

$$\begin{bmatrix} r_{h/e,N} \\ r_{h/e,E} \\ r_{h/e,D} \end{bmatrix} = A_{yaw}^{T} A_{pitch}^{T} A_{roll}^{T} \left\{ A_{sweep}^{T} \begin{bmatrix} m\sin\beta \\ m\cos\beta \\ 0 \end{bmatrix} + \begin{bmatrix} \delta x_{s/i,i} \\ \delta y_{s/i,i} \\ \delta z_{s/i,i} \end{bmatrix} \right\} + \begin{bmatrix} r_{i/e,N} \\ r_{i/e,E} \\ r_{i/e,D} \end{bmatrix} \tag{C.5}$$

Where: $\delta x_{s/i,i}$ is the offset of the SICK from the IMU along the x-axis in IMU co-ordinates (same notation for y and z axes)

APPENDIX D

There are multiple steps required to start the INS. There are four items that need to be powered up: the inverter, computer, GPS and IMU. Start the vehicle and switch on the inverter (make sure that the GPS and IMU are not powered at this time). After the inverter has started, turn on the computer and log onto one of the user names that has the C++ code for navigation.

The GPS requires a few steps in addition to being powered on. Plug the GPS power cord into the 12V accessory port (the green light on the plug will indicate when it is powered on). On the desktop, double-click the GPS_9600.hpt shortcut which will open com1. Wait 10 seconds before typing the following command:

com com1 57600

This command tells the GPS to switch from 9600 baud to 57600 baud. Every time the GPS is started, it reverts back to 9600 baud. It is necessary to communicate at 57600 baud to achieve 20 Hz strings from the GPS receiver. After typing the previous command in the HyperTerminal, close the window. Return to the desktop and double-click the GPS_57600.hpt shortcut. Type in the following command:

log com1 gpggalong ontime .05

This command tells the GPS receiver that the user wishes to log the gpggalong string (which includes latitude, longitude, height above sea level and a flag that indicates whether differential correction is available). The C++ code automatically connects to the GPS receiver after this point, but this step is required to verify that the GPS receiver is sending the gpggalong strings at 20 Hz. Also, OmniSTAR requires a few minutes for the differential correction to converge (which is indicated on the strings) for which the vehicle must not move. After the “W” on the gpggalong string, there will be a number indicating the accuracy of the solution. A “0” indicates that no solution is available because not enough satellites are available. A “1” indicates that the autonomous GPS solution is available (not corrected). A “2” or a “5” indicates that the GPS signal includes the correction from OmniSTAR. Once the GPS receiver has the corrected solution, close the window.

Drive the vehicle to the starting location which will be defined as the origin in the “main.cpp” C++ code. The vehicle will have a yaw angle that must be initially estimated. Open

the “RKF.sln” in Visual Studio.NET. At the top of the “main.cpp” code, there is a variable called “YAW_BEG”. Set the value of “YAW_BEG” as the initial estimate of the yaw angle of the AGV. Verify that the GPS is streaming at 57600 baud and that the IMU is not on.

Start the “RKF.sln” by hitting CTRL-F5. A command window will open waiting for further instruction. Run the “Controller.sln” program. Allow time for the connection to be established. Once both solutions are running, turn on the IMU by switching from 3V to 12V. The INS will now begin to send state information. The controller display will be visible when communication between all hardware is achieved. Once the pitch and roll angles have stabilized begin the test run.

To block GPS, make sure that the first command window is on top and hit “k” and then enter. “GPS blocked” will appear on the command window. In order to reacquire GPS, hit “k” and enter again. “GPS resumed” will now appear on the command window.

Once the test is concluded, close all command windows. The logs for the IMU, KF and Runge Kutta will now be available for viewing. The IMU log contains the accelerations and angular rates in units of g’s and °/s. The KF log contains all 21 states/parameters at 20 Hz including a flag that indicates GPS blockage (“0” for GPS available, “1” for GPS blocked). The Runge Kutta log contains the same information as the KF log (minus the flag) but at 71 Hz. If multiple runs are desired, make sure to change the names of the logs, otherwise they will be written over by the next run.

When the AGV is returned to the flight lab, turn off all the hardware. GPS needs to be unplugged otherwise it will drain the battery. The IMU just needs to be switched from 12V to 3V. Once the computer is shut down, the inverter can be turned off. Also, make sure to remove the batteries from the mouse.

APPENDIX E

The movie file included with this thesis is used to show the overall validity of the navigation solution for the Texas A&M AGV. The AGV (grey object) and SICK (blue object) locations are shown in the video for clarity. The AGV is traveling on the flat runway while logging the SICK returns and IMU measurements. The SICK hits can be seen as white dots and the ground is a flat, green plane with 1 meter squares.

VITA

Name:	Craig Allen Odom
Address:	4201 Hemlock St. Fort Worth, TX 76137-2020
Email Address:	craigodom@gmail.com
Education:	B.S., Mechanical Engineering, Texas A&M University, 2004 M.S., Mechanical Engineering, Texas A&M University, 2006